IN-61-CR
84331

Computer Science
Technical Report

P-94

Software Reliability Through Fault-Avoidance and Fault-Tolerance
Reports #5&6 (3/1/91-2/29/92) on NAG-1-983

by

Mladen A. Vouk and David F. McAllister

North Carolina State University

Box 8206
Raleigh, NC 27695

(NASA-CR-189880) SOFTWARE RELIABILITY THROUGH FAULT-AVOIDANCE AND FAULT-TOLERANCE Technical Report - 1 Mar. 1991 - 29 Feb. 1992 (North Carolina State Univ.) 94 p

N93-13293

Unclas

G3/61 0084331

Technical Report Submitted to the

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Langley Research Center, Hampton, Va.

for research entitled

SOFTWARE RELIABILITY THROUGH FAULT-AVOIDANCE AND FAULT-TOLERANCE

(Reports #5&6 on grant NAG-1-983)

from

Mladen A. Vouk, Principal Investigator, Associate Professor
David F. McAllister, Co-Principal Investigator, Professor

Department of Computer Science
North Carolina State University
Raleigh, N.C. 27695-8206
(919) 737-2858

Report Period
Beginning Date: March 1, 1991.
Ending Date: February 29, 1992.

Table of Contents

Summary of Accomplishments

This document is a synthesis of two semi-annual reports (#5 and #6). It covers the period from March 1, 1991 through February 29, 1992. The general topic of research was:

Strategies and Tools for Highly Dependable Software

Sub-topics were:

- Structure Based Testing, Reliability Growth, and Design Testability with Risk Evaluation. The work on this topic is in progress and some encouraging preliminary results are already available. Several reports and papers are available.
- Software Risk Management. Work on the topic is starting as far as highly dependable systems are concerned. However, some preliminary groundwork which will enable transition of the study to critical aerospace applications has already been undertaken. This preliminary work consists of studies relating to the general principles of software risk management, acquisition of tools, and analysis of reliability and availability of very large telecommunications systems.
- Software Fault-Tolerance. Studies in this area are in the process of being completed.

Structure based testing, reliability growth modeling, design testability, and risk evaluation

The objective of this work is to continue development of code coverage based reliability and test effectiveness models in order to improve fault-avoidance and fault-elimination during software production. These models relate the quality of the testing, as measured through metrics such as branch coverage, path coverage, definition-use pair coverage, etc., to the residual defect levels and reliability of the software, and therefore are intended to efficiently guide the testing process as well as offer insight into operational reliability of the product. An existing tool for computing different software code coverage measures is in the process of being extended to include new and promising metrics encompassed by the term "condition testing". A prototype is already available. The tool is being used to investigate RSDIMU software. The theory of coverage based testing has been extended to include more complex models. The models still need to be validated using experimental results from the RSDIMU test suite.

Break-down by sub-topic

* **Extended the BGG software tool for static and dynamic analysis of control and data flows in Pascal code to include reduced data-flow graphing and "condition testing" (or BRO) metrics.**

  The goal is to develop a sophisticated software tool for collection of complexity and execution coverage information on (RSDIMU) Pascal code.

  Accomplishments:
  The Basic Graph Generation and Analysis tool set (BGG), for dynamic and static analysis of Pascal code has been extended to allow analysis of reduced data-flow graphs and to include Tai's "condition based" testing measures.

Reduced data-flow graphs are useful in analyzing data-flow anomalies in the code. They are being used to study design testability issues.

Condition based testing focuses the testing process on predicates in a program. A new condition testing strategy called Boolean and Relational Operator (BRO) testing has been developed by Tai. To guide the test generation for BRO testing of a conditions, an algorithm developed by Tai was incorporated into BGG. We are in the process of using the tool to collect data on the relationship between the software errors discovered in RSDIMU software and the BRO coverage achieved during RSDIMU testing.

* **Formulated several new coverage-based reliability growth and test effectiveness models. The study of software reliability and availability models suitable for use during development of highly dependable software continues.**

The general goal of this part of the study is to provide additional theoretical and empirical basis for estimation of the reliability and availability of highly dependable software. Some of the problems associated with such models are their accuracy and the size of their predictive confidence bounds, and the inclusion into the models of the reliability of re-usable components and combinations of components developed at different sites with under different conditions.

Accomplishments:
The information collected with BGG has been used to formulate several preliminary coverage-based reliability model. Three types of error models are under current investigation: a linear model, an exponential model and a Rayleigh model. Preliminary results indicate that for a given coverage metric there is a lower bound on the reliability that can be achieved when the metric is fully satisfied. However, based on the currently available data it is not clear which of the models best describes the observed behavior of different metrics over a population of programs. Full coverage of hierarchically higher constructs (e.g. branch coverage is higher than statement coverage) would be expected to offer a better reliability of the final product. Additional experimental and theoretical work is needed.

* **Designing software for testability.**

The goal of the study is to provide empirical and theoretical information on fault avoidance and fault elimination properties of different coverage metrics with respect to software design for testability. Formal techniques will be used to identify critical software sections, and then design coverage metric mix that would provide some assurances concerning the level to which the code would being tested (e.g. parts of the code may be tested fulfilling branch coverage, while parts my need a more powerful metric such as definition-use paths). We believe that application specific design of metric mixes for the highest testability and sensitivity to high potential loss (risk sensitive) faults is particularly important when predicting behavior of highly dependable software. In addition, the study will provide theoretical and empirical basis for pre-release and operational phase estimation of the residual fault counts, reliability and availability of highly dependable software through combined coverage and time based models. Some of the problems associated with such models are the choice of the metric, the accuracy of point estimates, the size of the predictive confidence bounds, and possible inclusion into the models of the reliability of re-usable components and combinations of components developed at different sites under different conditions.

Accomplishments:
Research is still in progress. Currently BGG extensions that would enable computation of appropriate metrics are not fully implemented. The SDT CASE tool based on the Software Description Languages for development of real-time systems is on order and is expected to be delivered sometime this summer. The tools allows expression of software designs in therms of a special design language. Code generation is automatic. The code would be analyzed via BGG and design structures which generate most testable code would be sought. Also in progress is development of a Software Risk Management system which would coordinate software risk management activities associated with the development of critical products. It is expected that a fault-tree analysis tool would be combined with SDT and BGG outputs to attempt fault-tree retroactive analysis in order to identify safest design strategies and code generation approaches.

* **Papers and reports. Several preliminary reports are available. Two papers are in preparation.**

The tool set has been described in a conference paper. BGG extensions in the area of are described in Appendix II. Journal paper describing the tool set and its theoretical underpinnings (particularly some new metrics such as control and data-flow based definition-use-redefinition chains, BRO metrics, etc.) is in preparation. A paper on coverage based reliability models will be presented in June 1992 at the Quality and Productivity Research Conference (Corning, NY). Journal paper is in preparation.

1. Vouk, M.A., and Coyle, R.E., "BGG: A Testing Coverage Tool," Proc. Seventh Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Inc., Portland, OR, pp212-233, September 1989.
2. Borger D, "BGG User's Manual", NCSU Department of Computer Science, 1990. (available on request)
3. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, May 16, 1990.
4. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, January 15, 1991.
5. Vouk, M.A. and Tai, K.C "Software Testing and Reliability", Summary of the Presentation Prepared for the Workshop on Issues in Software Reliability Estimation, Purdue University, May 21, 1991 **(Appendix I)**
6. K.C. Tai, "Theory of Condition-Based Software Testing", Draft Paper, September 1991 (available on request)
7. Vouk, M.A., Tai, K.C., Staats W., Koorapathy H. and O'Connor, J., "Extensions to BGG Testing Coverage Tool," report in preparation **(Appendix II)**.
8. Vouk, M.A., "Modeling Software Reliability and Fault Removal During Structure Based Testing," 9th Quality and Productivity Research Conference, Corning, New York, June 1992 (paper accepted, final version in preparation)

Reliability growth models and software risk management

* **Reliability and availability models suitable for use during development of highly dependable software-based systems are being developed and evaluated.**

The goal is to provide additional theoretical and empirical basis for estimation of the reliability and availability of highly dependable software. These models include coverage based and time-based models. Some of the problems associated with such models are their

accuracy and the size of their predictive confidence bounds, and the inclusion into the models of the reliability of re-usable components and combinations of components developed at different sites with under different conditions. Expected future accomplishments include: extended software testability modeling based on control and data flow construct coverage, preliminary model of a multi-component (re-use and build) reliability and availability, and formulation of a reliability models driven by specification and design error analysis, so that problem areas can be identified prior to software implementation.

Accomplishments:
Reliability and availability models suitable for use with very large critical multi-component telecommunications systems are being studied. Particular attention is directed at multi-state non-homogeneous Markov and semi-Markov models which can be used to account for a variety of system failure types, as well as for hardware/software interaction. The knowledge gained will be used in building appropriate models for the highly-dependable aerospace applications.

* **Software process and risk management model appropriate for development of very large and complex critical software systems is being studied.**

The goal of this part of the study is to extend theoretical and empirical basis for risk management of highly dependable software. Some of the problems associated with such models are their predictive accuracy, the inclusion into the models of the risks associated with re-usable components and combinations of components developed at different sites under different conditions. Existing risk based software development models such as the Spiral Model will be evaluated and if possible supplemented with state of the art reliability models. Expected future accomplishments include: incorporation of coverage based software reliability models into existing risk models and development of a new or extended models, incorporation of existing and new multi-component (re-use and build) reliability and availability models into a risk-driven development model for highly dependable software, investigation of predictive properties of the existing and new risk models drive by requirements and design change and error reports.

Accomplishments:
The models for software process and risk management models are being reviewed for relevance to highly dependable systems. A research prototype of a software process and risk management system is under constructions. The system is expected to provide guidance for risk-based process and software design and provide risk evaluation tools such as software reliability and availability estimation modeling, fault-tree analysis, schedule analysis and statistical decision making.

* **Reports and papers in preparation:**

1. D.S. Borger. M.A. Vouk, "Modeling the Behavior of Large Software Projects", NCSU Center for Communications and Signal Processing, Technical Report TR-91/19, June 91.
2. M.A. Vouk, Proc. "Engineering of Telecommunications Software", TRICOM '92, pp 281-296, February 1992.
2. R. Cramp, M.A. Vouk, W. Jones, "On Operational Availability of a Large Software-Based Telecommunications System", submitted to ISSRE92.

Evaluation of Consensus Voting, Consensus Recovery Block, and Acceptance Voting

The objective of this study is to investigate advanced software fault-tolerance models in order to provide alternatives and improvements in situations where simple software fault-tolerance strategies break-down. One such situation is the presence of significant inter-version failure correlation which, for example, makes reliable voter operation problematic. We have evaluated an improved voting strategy called Consensus Voting (CV) which automatically adapts to different version reliability and output space cardinality characteristics. CV reliability performance is always as good or better than that of majority voting. We have also evaluated performance of Consensus Recovery Block (CRB) and Acceptance Voting (AV) models. From the cost and reliability perspective the CRB model is superior to all investigated stand-alone voting schemes, even in the presence of failure correlation. The AV scheme reduces, or completely eliminates, as many wrong answers as possible by acceptance testing them before dynamically voting on them. AV is very dependent on the reliability of the acceptance test, but under special circumstances AV reliability and safety performance can be better than that of CRB or CV.

Break-down by sub-topic

* **The reliability performance of Consensus Voting (CV) was validated using RSDIMU software. From the reliability perspective CV is superior to majority voting as a stand-alone voting technique .**

  The goal of the study was to validate effectiveness of Consensus Voting, a technique theoretically shown to posses considerable positive auto-adaptive properties in small output spaces and in the presence of highly correlated failures.

  Accomplishments:
  a) Analyses confirm the theoretical and simulation results that the net effect of failure correlation is to change the size of the output space in which a voter makes decisions.
  b) Consensus Voting (CV) may in part compensate for the problems that otherwise arise in the presence of failure correlation with classical voting strategies such as Majority Voting. Consensus voting automatically adapts to different component reliability and output space cardinality characteristics.
  c) Theory prediction that in small output spaces CV performs as well or better than majority voting, while in large output spaces its performance compares with 2-out-of-n voting was confirmed. This was confirmed using RSDIMU programs.

* **The reliability and safety performance of Consensus Recovery Block (CRB) was validated using RSDIMU software. From the reliability perspective CRB is superior to any stand-alone voting technique.**

  The goal of the study was to validate effectiveness of Consensus Recovery Block, a technique theoretically shown to superior to simple voting.

  Accomplishments:
  a) Analyses confirm the theoretical and simulation results that CRB is a superior technique from the standpoint of reliability.
  b) CRB remains superior even in the presence of considerable inter-version failure correlation, although attention must be paid to the choice of the strategy used in the voting stage (e.g. in some cases CRB with MV front end performs better than CRB with CV).

c) CRB safety properties are good with high reliability versions, but are inferior to Acceptance Voting, or even simple Recovery Block strategies, for medium to low reliability versions.

* **The reliability and safety performance of Acceptance Voting (AV) was validated using RSDIMU software. Under special circumstances AV reliability and safety performance can be better than that of CRB or CV.**

The goal of the study was to validate effectiveness of Acceptance Voting (AV), a technique designed to compensate for some problems that arise with CRB and CV in very small output spaces and under very high failure correlation.

Accomplishments:

a) Analyses confirm the theoretical and simulation results that under special circumstances AV reliability performance can be better than that of CRB, or CV. For example, in binary output space, with N >5, there are regions of version reliability where AV gives better system reliabilities than CRB or CV models.
b) In general, AV reliability performance is inferior to CRB and is very dependent on the quality of its acceptance test stage.
c) In the case of medium to low version reliabilities, AV safety properties tend to be better with than those of CRB under any voting strategy. However, when AV acceptance test reliability is low, or version reliabilities are very high, AV may perform worse than CRB. It usually remains safer than plain Recovery Block.

* **The cost of Consensus Recovery Block (CRB), Recovery Block (RB) and N-Version Programming was studied. Result indicate that unless the voter is perfect, N-Version Programming does not compete cost-wise with the other two methods. Given failure independence CRB is superior to RB.**

The goal of the study was to study cost-effectiveness of more common fault-tolerance strategies in situations where interversion failure correlation is negligible.

Accomplishments:

a) In the case of failure independence Consensus Recovery Block and Recovery Block are the only cost justifiable fault-tolerant techniques to be considered. Unless the voter is perfect, N-Version Programming does not compete cost-wise with the other two methods.
b) However, the hybrid method Consensus Recovery Block which contains both voting and recovery block can provide considerable reduction in cost for a given system reliability over the other techniques.

* **Papers and reports.**

1. Athavale A., "Performance evaluation of hybrid voting schemes", North Carolina State University, Department of Computer Science, M.S. Thesis, December 1989.
2. M.A. Vouk, and D.F. McAllister, "Preliminary Report on Consensus Voting in the Presence of Failure Correlation" in *Software Reliability Through Fault-Avoidance and Fault-Tolerance*, NASA grant NAG-1-983 Progress Report #2 (9/1/89-3/31/90), 1990.
3. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, May 16, 1990.

4. D.F. McAllister and R. Scott, "Cost Modeling of Fault Tolerant Software", Information and Software Technology, Vol 33 (8), pp 594-603, October 1991 **(Appendix IV)**
5. M.A. Vouk, D. F. McAllister, D.E. Eckhardt, and K. Kim, "An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the Presence of Failure Correlation," submitted to the Special Issue of Journal of Computer and Software Engineering, March 1992, **(Appendix III)**

Appendix I

Summary of the Presentation Prepared for the Workshop on Issues in Software Reliability Estimation Purdue University, May 21, 1991

Software Reliability and Testing*

Mladen A. Vouk
North Carolina State University
Department of Computer Science

K.C. Tai
National Science Foundation
Washington, D.C.

* Research supported in part by NASA Grant No. NAG-1-983

Introduction

Software testing, complexity and reliability are three software research subjects that have been studied for some time. However, there appears to be a lack of study on the relationship among these subjects. We feel that more research on this relationship is needed and that such research will lead to significant results for increasing software quality. In the presentation we described a simple model that relates the code coverage and the error removal process. The model was verified using a set of functionally equivalent programs. We also offered a model that relates software reliability and coverage. Its verification is in progress. The described models provide a strong argument in favor of judicious use of different control and data-flow metrics during testing, but the results also indicate that considerable caution and understanding is needed when interpreting attained coverage values.

2. Software Complexity and Testing

A number of program complexity metrics have been defined and many of them are directly or indirectly related to testing strategies. Below we show examples of the correspondence between complexity metrics and testing strategies:

complexity metrics		**testing strategies**
number of statements	<=>	statement testing
control-flow based metrics	<=>	control-flow based testing
data-flow based metrics	<=>	data-flow based testing

Currently it is not clear which metric(s) can best represent the complexity of a program, and indeed is one single metric sufficient for this task. It is also not clear which testing strategy or strategies are most effective for error detection. One consideration for the complexity of a program is how the input domain of the program is partitioned. We are currently studying the relationship between the conditions in a program and the partitions in the program's input domain. And we are using this relationship as the theoretical basis for the condition testing approach.

3. Software Testing and Reliability

The reliability of a program is based on the results of testing. Existing reliability models can be are based the use of random testing. The book Software Reliability by Musa, Iannino and Okumoto include the following statements:

(1) "..., for accurate reliability measurement during test, select runs

randomly with the same probability expected to occur in operation."
(2) "... that the input space must be well covered for accurate reliability measurement."

The major goal of research on non-random testing is to select tests that are effective for error detection. The choice of a test criterion for a program affects the effectiveness of testing this program and thus the reliability of this program. We have the following two hypotheses:

Hypothesis 1: If we apply a more effective testing strategy to test a program, then the resulting program (after corrections have been made) is be more reliable.

Hypothesis 2: The reliability of a program is a function of the program's complexity, the test criterion to be satisfied, and the errors detected during testing.

Existing reliability models, however, are not based on the above two hypotheses; they use random testing for both the prediction and measurement of reliability. We feel that although the reliability of a program is measured by using random testing, it could be predicted by using non-random testing. Therefore the following questions arise:

(1) Is random testing as effective as non-random testing for error detection?

Should both random and non-random testing be used for error detection?

(2) How to predict the reliability of a program based on the results of non-random testing (with or without random testing)?

For the first question, conflicting research results have been reported. The book by Musa et al. mentioned the results of Curtis and Durant and concluded that random testing performs well with respect to branch and path testing. However, our research results conflict with this conclusion.

4. Some S/W Testing and Reliability Research Issues Considered at NCSU

- Multi-dimensional View: Structure Based S/W Testing
  - Metrics (e.g. Condition based testing, dud-chains, data and control-flow density)
  - Error Removal and Reliability Models
  - Random vs. Functional Testing

- Operational S/W Reliability and Availability
  - Commercial software with very large distribution (1,000 - 100,000 systems)
  - Early Identification of Operational Profiles

- Special Development and Testing Approaches
  - Parallel Component Testing (application of Markov chain models)
  - Back-to-Back Testing
  - Releases vs. Continuous Process

- Reliability of Software Fault-Tolerance Mechanisms
  - Hybrid models

5. Conclusion

We propose that the relationship among software testing, complexity and reliability be carefully studied. This study is expected to produce useful results for improving the quality of computer software. Some immediate research and development goals

- Investigate (in a quantitative terms) the place, role and value of structure based testing in the software process.
- Provide theoretical foundation and experimental information on the costs and efficiency of structure based testing and reliability growth monitoring
  - for software quality forecasting
  - test stopping criteria
  - test direction criteria (diminishing returns, testing efficiency saturation)
  - selection of error-sensitive test cases
  - accelerated testing
- Tools
- Design for Testability and Reliability

Workshop on Issues in Software Reliability Estimation
Purdue University

Software Reliability and Testing

Mladen A. Vouk
North Carolina State University
Department of Computer Science

West Lafayette, IN, May 21, 1991

Some Research Issues Considered at NCSU

- **Multi-dimensional View: Structure Based S/W Testing**
  - Metrics
  - Error Removal and Reliability Models
  - Random vs. Functional Testing
- **Operational S/W Reliability and Availability**
  - Commercial software with very large distribution (1,000 - 100,000 systems)
  - Early Identification of Operational Profiles
- **Special Development and Testing Approaches**
  - Parallel Component Testing (application of Markov chain model)
  - Back-to-Back Testing
  - Releases vs. Continuous Process
- **Reliability of Software Fault-Tolerance Mechanisms**
  - Hybrid models

Personnel

- **NCSU Computer Science Faculty - Software Systems**
  - Dr. Rance Cleaveland
  - Dr. David McAllister
  - Dr. K.C. Tai (currently with NSF)
  - Dr. Mladen Vouk
- **Some current graduate students and their research topics (Software Testing and Reliability)**
  * Mr. Young Choe (Ph.D.) - testing strategies of distributed systems and protocols.
  * Mr. Randy Cramp (Ph.D.) - software reliability, availability, and testing strategies.
  * Mr. Kim Kalhee (Ph.D.) - software reliability and testing, back-to-back testing.
  * Mr. Garrison Kenney (Ph.D.) - software reliability.
  * Mr. Wayne Staats (Ph.D.) - data-flow analysis, structure based testing, software errors.
  * Mr. Robert Coyle (M.S. with thesis) – software structure based testing tools.
  * Mr. Dana Borger (M.S. with thesis) – software process modeling, testing strategies and tools.
  * Mr. Satya Vemulakonda (M.S. with thesis) – multiversion failure correlation effects, errors and faults.

Other students, and researchers from other universities and industry have been contributing through participation in other projects and cooperative research efforts.

Overview

Structure Based S/W Testing

- New Metrics: Condition Testing (K.C. Tai), DUD-chains, data and control flow density (M. Vouk, K.C. Tai)

- Reliability Models: Error Removal and Reliability Models (M. Vouk, K.C. Tai)

- Testing Strategies: Random vs. Functional Test Data (M. Vouk and K.C. Tai)

Operational S/W Reliability and Availability

- Estimating Defects in Commercial Software During Operational Use, G. Kenney (M.S. Thesis, NCSU, 1991)

- Two commercial software systems were investigates. One deployed at 10,000 and the other at 100,000 hosts, no record of CPU time, field reported defect count is the primary driver.

- A Weibull model appears to describe the process quite well.

$$\lambda_u = N\alpha\beta t^{(\beta-1)}e^{-\alpha t^{\beta}}$$

unique failure rate, total number of defects, failure rate decay per system, system usage grow rate.

Special Development and Testing Approaches: A Custom-Designed Reliability Model for a Stable Software Testing Process

- Process & system (based on Work of Dr. C. Wholin, University of Lund, Sweden, 1991

- Violation of standard assumptions, lack of adequate models.
  - independence of failures
  - testing profile
  - immediate correction
  - decreasing failure rate
  - testing load

- A more detailed Markov chain model is being developed.

Structure Based Testing: Motivation

Some (Testing) Problems

- Relating reliability to the overall software development process, environment and participants (developers, users, maintainers) - multidimensional nature of the problem.
- A prescriptive, or proscriptive models.
- How reliable is the software: on release and in operation?
- How robust is this reliability?
- What is the quality of testing? When to stop testing?
- Is testing process moving in the right direction? Is it adequate?
- How to (cost-efficiently) choose error-sensitive test cases?

  etc.
- Problems are particularly acute for complex software systems that need to be highly dependable.

Some Possible Solutions

- Measure adequacy of the testing effort while it is taking place. For example, functional and structural coverage metrics.
- Measure software reliability growth and estimate residual fault count. Test "exposure time" in several dimensions (e.g. CPU execution time, calendar time, structural exposure coverage).
- Guide testing by determining when diminishing returns set in, and a change in strategy is required.
- Relate employed reliability models to the stability of the development and testing process.
- Formulate prescriptive development and testing models that guarantee certain lower bound on reliability.

Software Error Removal and Reliability Modeling

- "Classical" Failure Intensity vs. Execution Time based models are in current use in software industry.

- They model the debugging process based on the requirement that an operational profile be used.

- Some of the problems associated with such models are
  - Use of true operational profile may translate into an inordinate number of test cases (time) to test the system statistically.
  - Operational profile may not be known.
  - Robustness of the estimates to changes in the development and testing processes as well as operational environment changes is usually not considered.
  - Accuracy, size of their predictive confidence bounds.
  - Inclusion into the models of the reliability of re-usable components and combinations of components developed at different sites with under different conditions, etc.
  - Incremental development, releases, etc.

- Alternative "exposure time" metrics, that would also guide selection of test cases, may offer complementary information and improve.

Coverage

- Coverage C(M,S) is computed for a construct quantified through metric (M) and testing strategy (S).

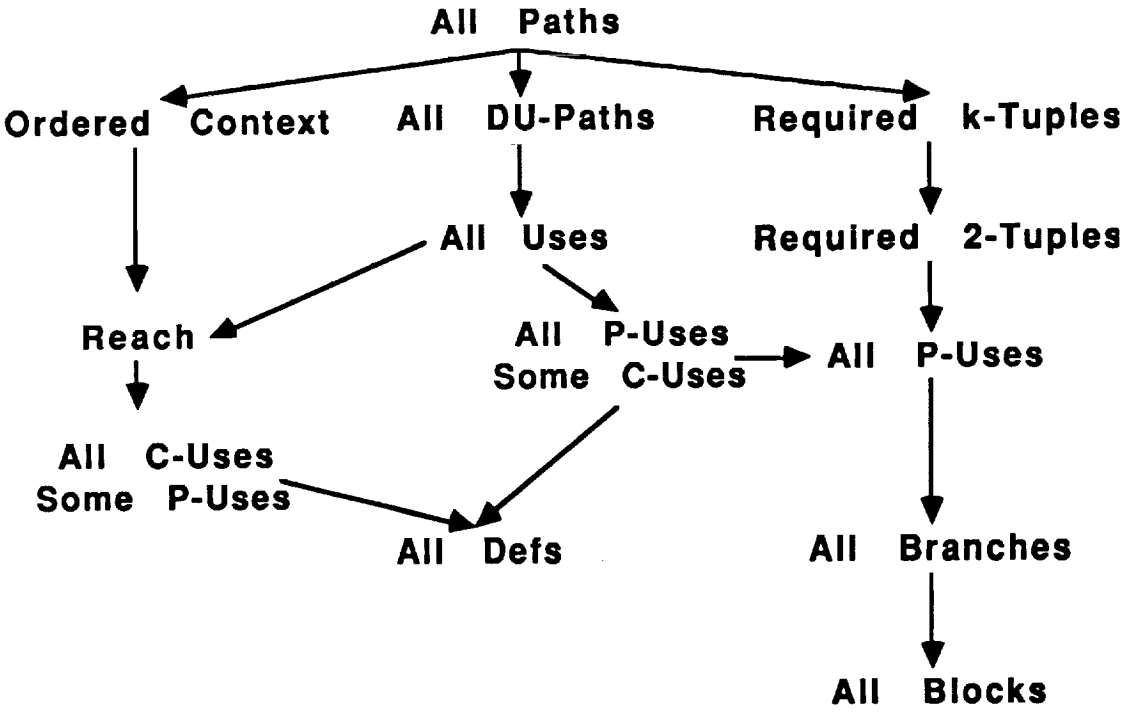
$$C(M,S) = \frac{\text{No. of executed constructs for M under S}}{\text{Total no. of } \textit{executable} \text{ constructs for M under S}}$$

- Examples of Unit Testing Oriented Control Flow Metrics:
  - Lines of Code (executable)
  - Linear blocks of code
  - Branches
  - Partial Paths (linear-code-and-jump, LCAJ)
  - Paths

- Examples of Unit Testing Oriented Data Flow Metrics:
  - All definitions (in predicates and linear blocks)
  - All uses (in predicates and linear blocks)
  - All definition-use tuples
  - All definition-use paths

- A typical hierarchy of control and data flow based measures is the following one [Cla90]. The aim is to indicate which of the criteria subsumes which other criteria, and by implication which of the measures may be superior as a practical complexity or coverage measure.

- Sheer number of paths does not necessarily qualify. Even the simplest loop can result in an infinity of paths.

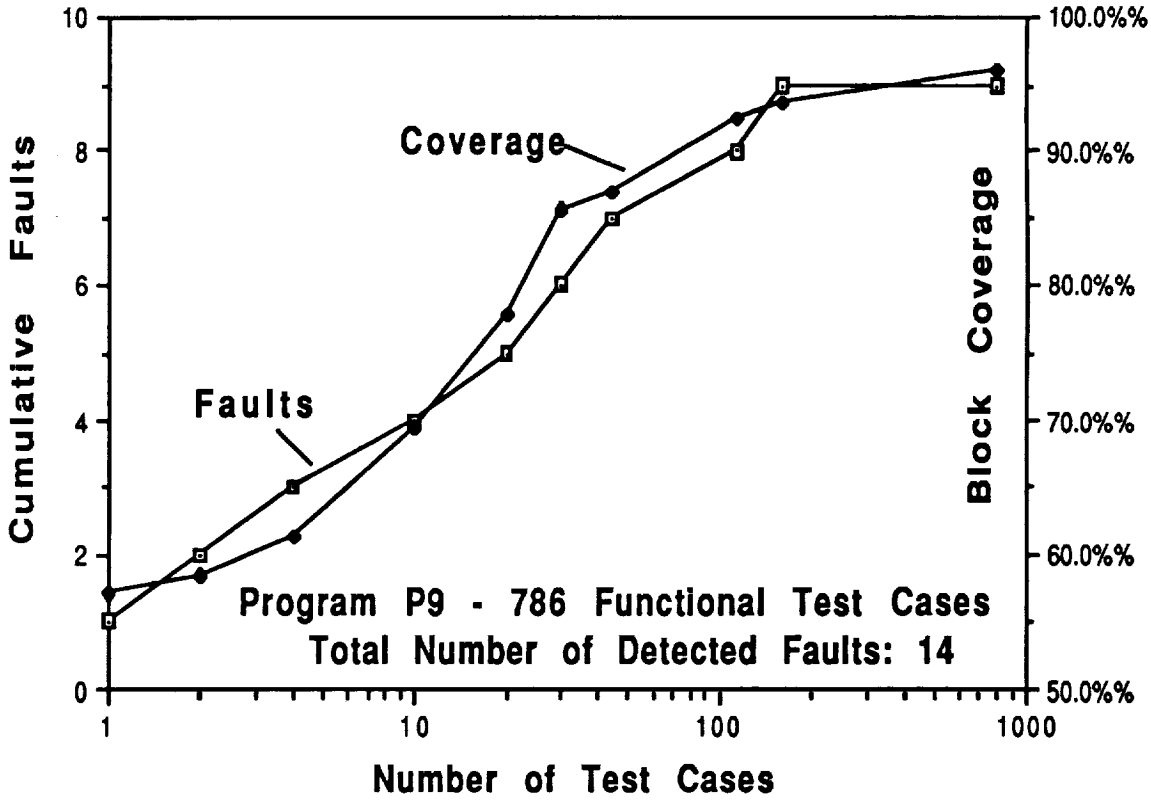


Tools

- We have developed a tool (BGG) that handles static and dynamic analysis of control and data flow graphs (global, inter-, and intra-procedural data flow) for program units written in full Pascal (unit is considered to be about 4,000 lines of code).
  - Examples of computed static measures: branches, paths, cyclomatic number, definition-use (du) pair counts, count of definition-use paths, average definition-use path lengths, p-uses, c-uses, and all-uses.
  - Dynamic coverage is computed for definition-use pairs, definition-use-redefinition chains, p-uses, c-uses and all-uses.
- The tool is used to analyze different sets of programs.

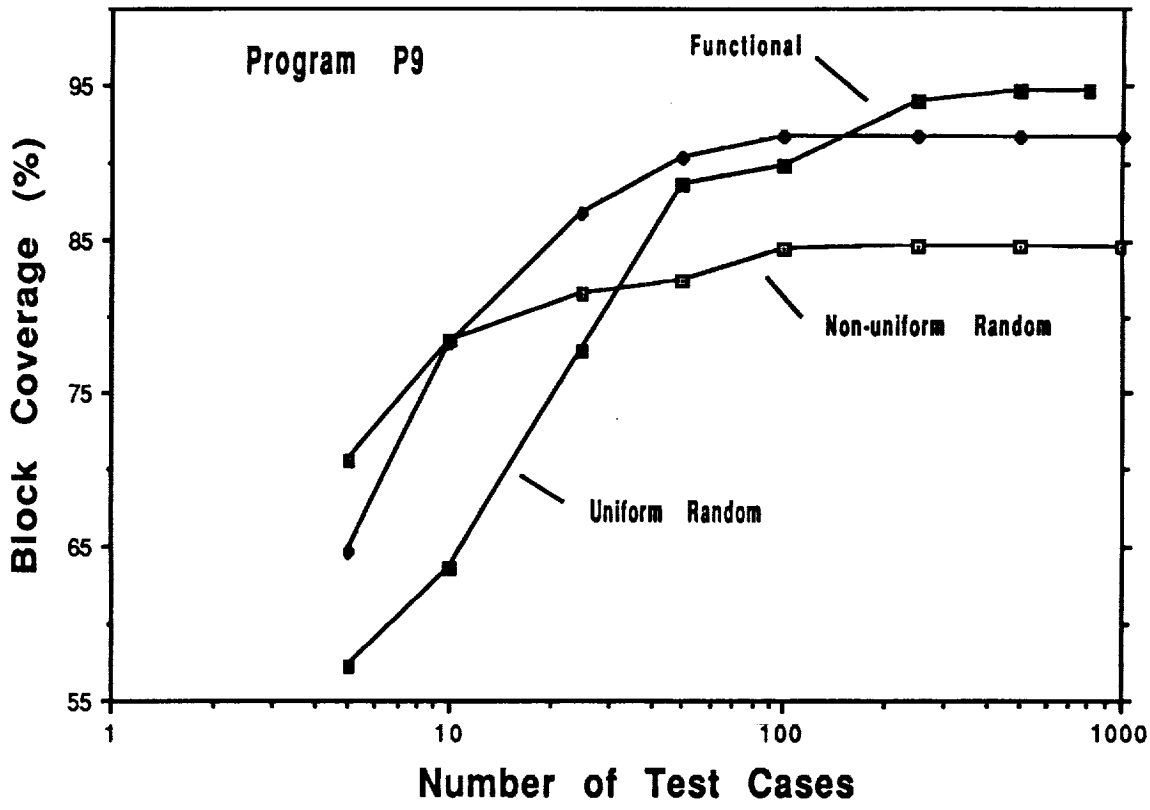
Some Motivating Results

- **Increased code coverage brings about increased fault-detection.**



Growth of block coverage and of the number of detected and corrected faults with the growth in the number of executed functional test cases.

- **The coverage provided by random data is very dependent on profile of the test data, and use of extremal and special value test cases usually provides a better results and detects more faults.**



Comparison of linear block coverage observed for two random testing profiles and a functional data set with a Pascal program.

- There appears to be considerable variability in the meaning a given coverage C for metric M caries over a population of functionally equivalent programs

- For instance, the same set of test cases may not provide the same level of coverage over all functionally equivalent software, and that for exactly the same test data sets coverage could vary as much as 22% across the population of functionally equivalent programs.

- All indications are
  - that coverage has very limited meaning as a stand alone measure,
  - that coverage "exposure" should be directly associated with software reliability (quality) growth, and
  - that appropriate models relating development and testing strategies, coverage, and reliability need to be investigated.

- Several structure based testing error removal models are under investigation.

- A two-dimensional structure and time based reliability model is under development.

- Results are encouraging, but confirm that considerable caution is needed in the interpretation of coverage measures.

Structure Based Testing: Some Immediate Goals

- Investigate (in a quantitative terms) the place, role and value of structure based testing in the software process.
- Provide theoretical foundation and experimental information on the costs and efficiency of structure based testing and reliability growth monitoring
  - for software quality forecasting
  - test stopping criteria
  - test direction criteria (diminishing returns, testing efficiency saturation)
  - selection of error-sensitive test cases
  - accelerated testing
- Tools
- Design for Testability and Reliability

Structure Based Testing: Error Removal Models

- Rayleigh Model
- Exponential Model
- Linear Model

- **Some simplified assumptions about the coverage based testing and fault removal process:**

  (1) Coverage based testing and fault removal, in the first approximation, is equivalent to sampling without replacement.

  As the result: fault detection rate is proportional to the coverage, $1 \geq C(M,S) \geq 0$.

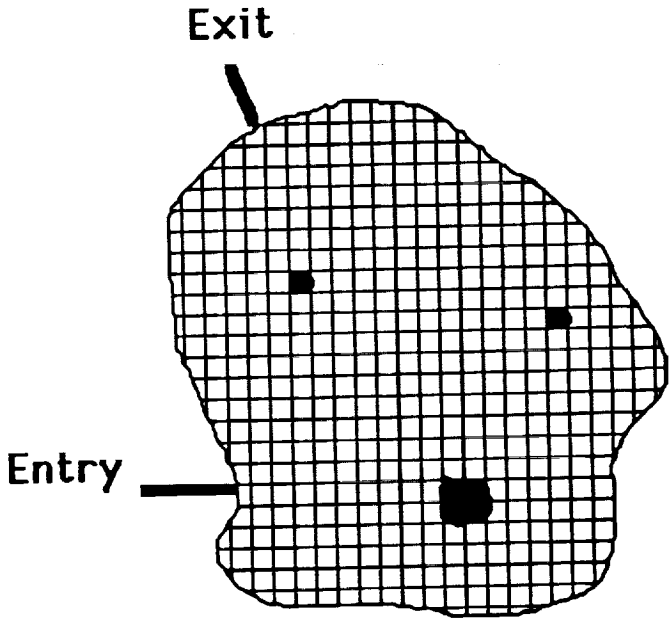
  To increase (fulfill) coverage we generate test cases which would cover as many yet uncovered constructs as possible. "Re-use" of constructs through new paths that exercise as at least one new construct is "ignored" by the metric M.

  Order of execution of test cases is ignored (and is assumed random) unless otherwise dictated by the testing strategy S.

  (2) The rate of fault detection is proportional to the number (or density) of residual faults, ε_r, detectable by metric M under strategy S.

  (3) For each test set T generated under S and monitored through M, there is a minimal coverage and maximal coverage achievable.

M-space of program P



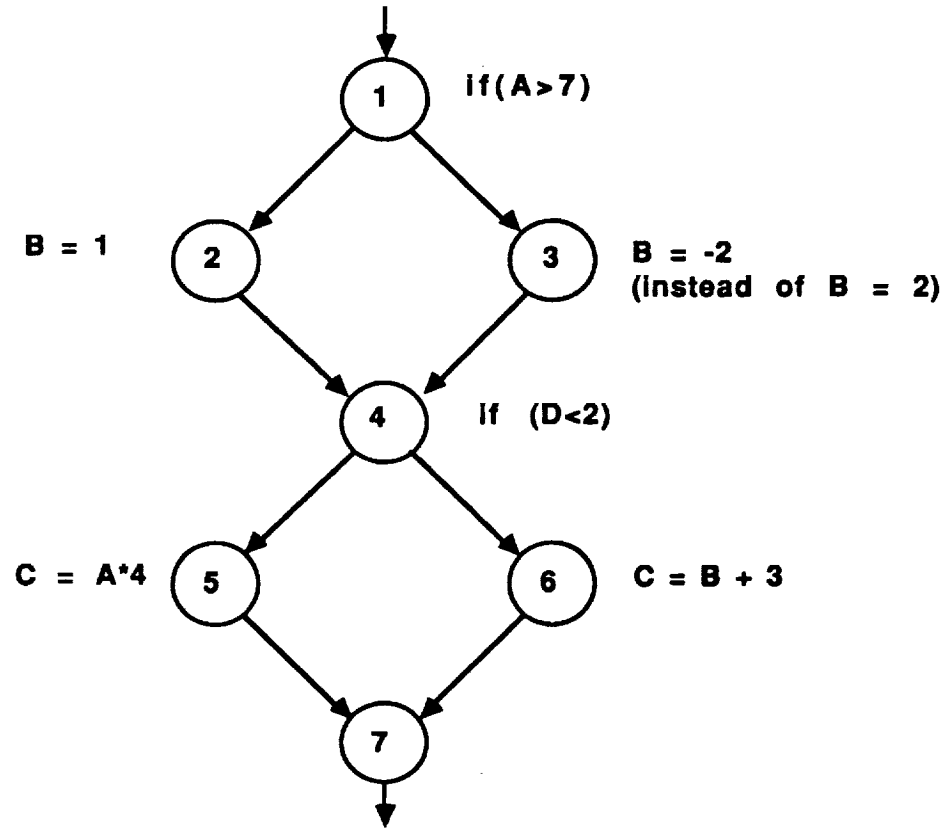
- As the number of <u>unexecuted</u> M constructs shrinks the probability increases of "trapping" a M-detectable fault that remains.
- Each fault, X, has associated with it probability p(M,S,X) that it is detectable by M under S. So

$$\varepsilon_r = \sum_{i=1}^{E_r} p(M,S,X_i)$$

where E_r is the actual number of faults remaining.

- The number of faults (initial, detected, remaining, etc) could be normalized over the total number of M constructs in program P (density).

Probability of Detecting X by M under S



- Strategy: Cover all branches at least once.
- Full branch coverage can be achieved in many ways. For example via paths:

  (1-2-4-5-7, 1-3-4-6-7), or (1-2-4-6-7, 1-2-4-5-7), or (1-2-4-5-7, 1-2-4-6-7, 1-3-4-5-7), etc.
- Only some of these path combinations detect the error.

- The fault detection rate with respect to coverage is

$$\frac{d\varepsilon_d}{dC} = k\ \varepsilon_r\ C$$

where ε_d is the number (or density) of detected faults, and C is C(M,S).

- Under a simplifying assumption that fault correction is instantaneous and perfect (i.e., no fault generation, $\varepsilon_g = 0$), the number of corrected faults, ε_c, is equal to number of detect faults.

- Number (or density) of residual faults is

$$\varepsilon_r = \varepsilon_T - \varepsilon_c$$

where ε_T is the total (effective) number of (M-detectable) faults in program P at coverage C=0.

- Hence

$$\frac{d\varepsilon_c}{dC} = k\ (\varepsilon_T - \varepsilon_c)\ C$$

- **Solution of this differential equation in the range $0 \leq C \leq 1$, and initial condition $\varepsilon_c = 0$ for $C \leq C_{min}$**

$$\frac{d\varepsilon_c}{(\varepsilon_T - \varepsilon_c)} = k\, C\, dC$$

yields

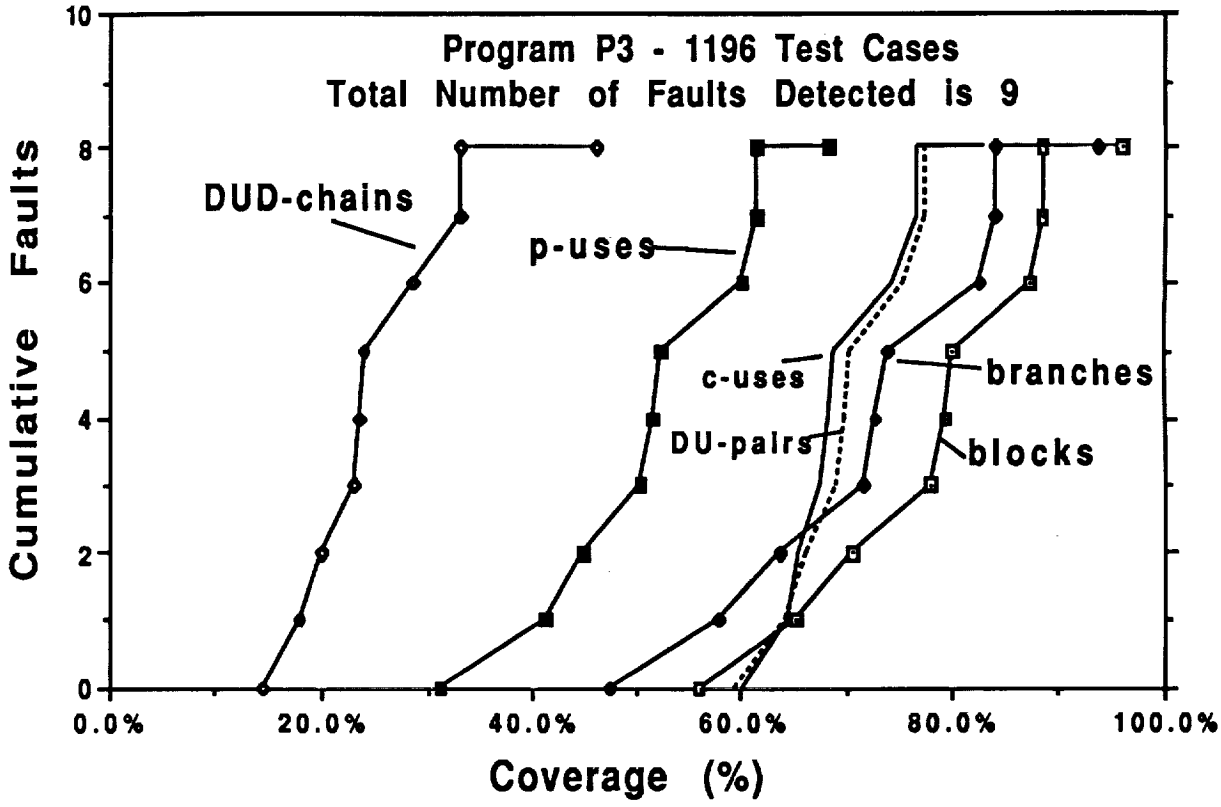
$$\ln\left(1 - \frac{\varepsilon_c}{\varepsilon_T}\right) = -\frac{1}{2} k\, (C^2 - C_{min}^2)$$

or

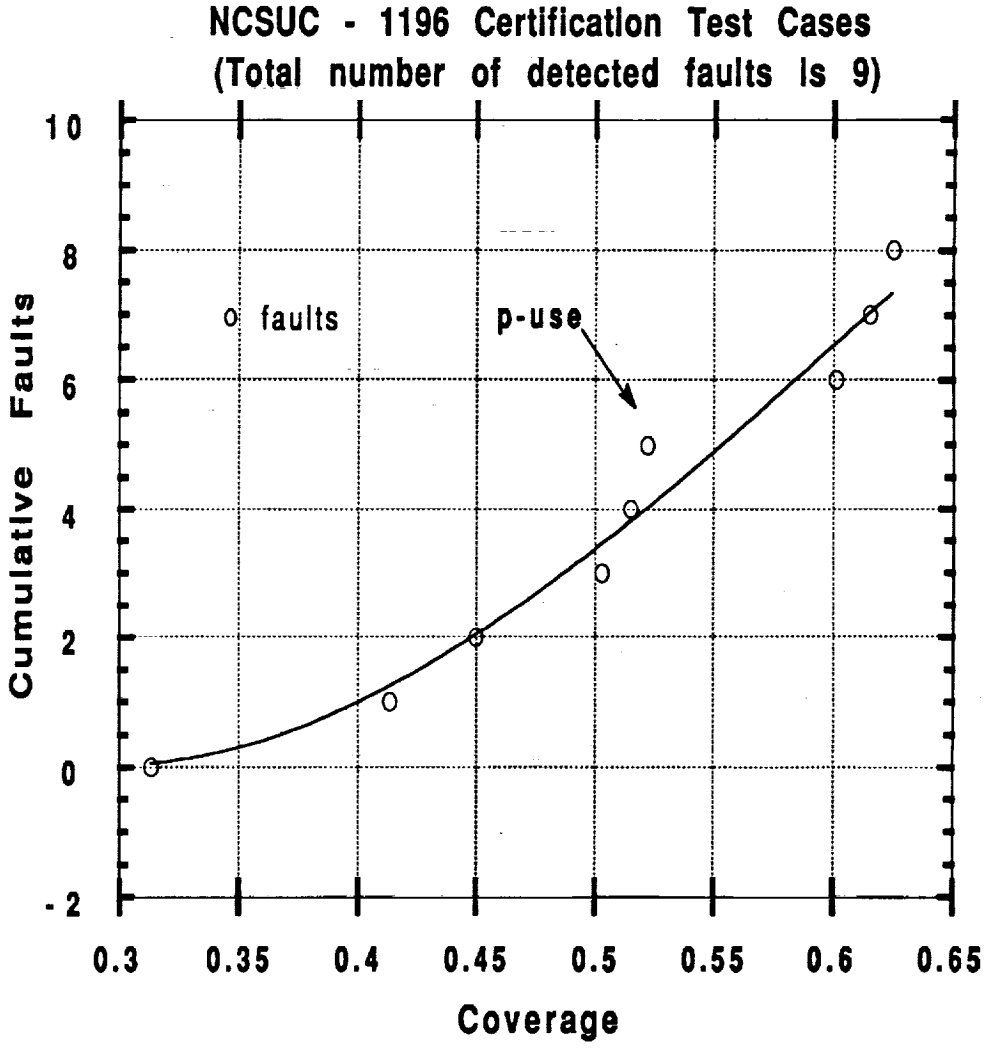
$$\varepsilon_c = \varepsilon_T[1 - e^{-\beta(C^2 - C_{min}^2)}]$$

- **This is a variant of the Rayleigh distribution, i.e. a special case of the Weibull distribution.**

Examples of Experimental Results

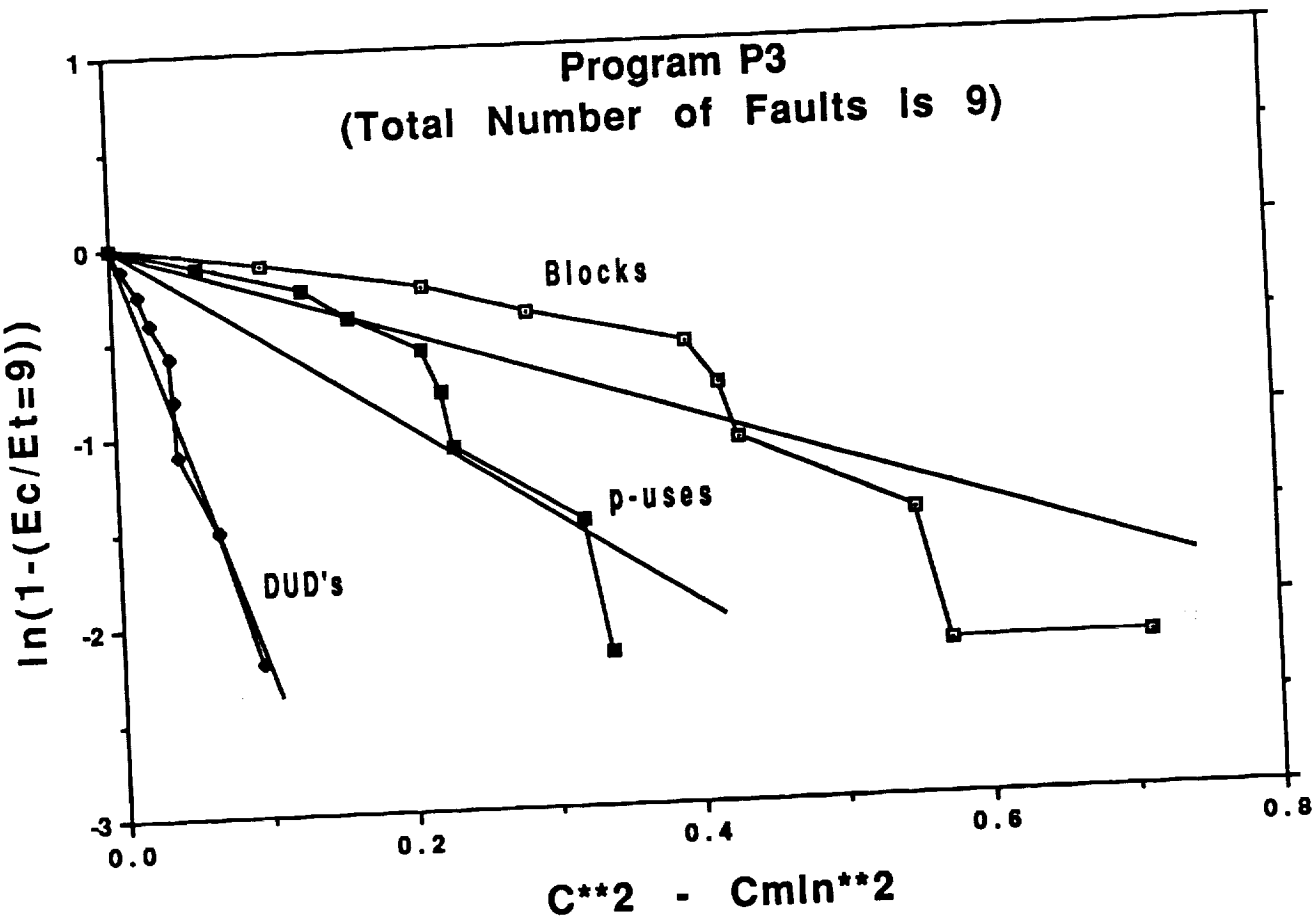


Direct Fit of $\varepsilon_c = \varepsilon_T[1 - e^{-\beta(C^2-C_{min}^2)}]$



Power of Coverage Metrics

$$\frac{\varepsilon_c(C=1)}{\varepsilon_T} \leq 1 - e^{-\beta(1-C^2_{min})}$$



- The sharper the slope, β, and the smaller C_{min}, the higher is the potential of the metric in detecting faults.

Block : p-use : DUD-chains $\approx$ 1 : 2 : 6

Fault Detection Properties of Random and Functional Testing

- **In our experiments most of the time random data showed poorer overall fault detection properties than functional data. However, random testing did uncover a certain number of faults not detected by functional testing.**

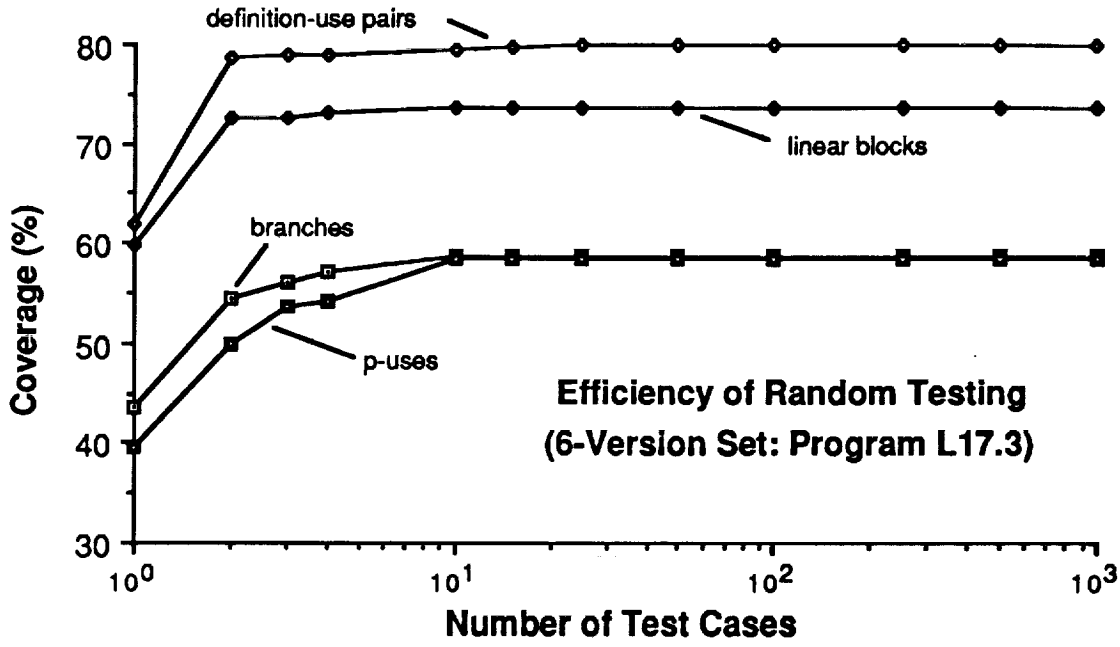
Test Set	Fault Type Dissimilar	Similar	Total
Random+ESV-I	54	7	61
Random	48	2	50
ESV	48	7	55
Random but not ESV	6	0	6
ESV but not Random	6	5	11

The number of faults detected by ESV and 500 RANDOM-Ib cases.

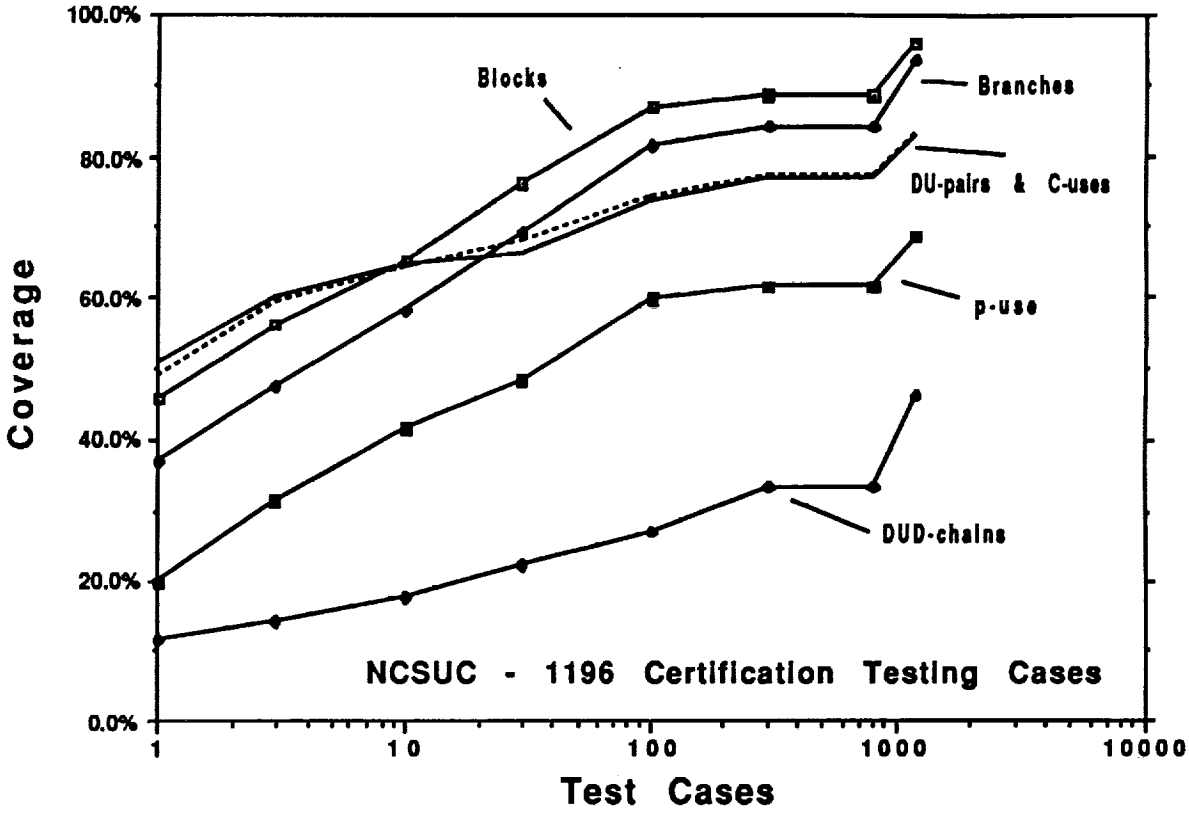
Actual	Program	400 ESV-I	500 RANDOM-Ib	Total
ncsua	P1	4	2	4
ncsub	P2	5	1	6
ncsuc	P3	7	3	8
ncsud	P4	9	3	11
ncsue	P5	8	3	8
uclaa	P6	5	3	5
uclab	P7	3	1	4
uclac	P8	6	4	7
uclad	P9	9	3 - 5	10
uclae	P10	3	1	4
uiuca	P11	4	3	4
uiucb	P12	4+	1+	4+
uiucc	P13	6	3	9
uiucd	P14	6	5	6
uiuce	P15	5	3+	6
uvaa	P16	6	1	6
uvab	P17	5	4	6
uvac	P18	5	1	7
uvad	P19	1+,(6)?	1	6
uvae	P20	4	2	5

Coverage by Random and Functional Testing

- **Our experiments show that as many as 40% of the branches could be missed by random test data.**



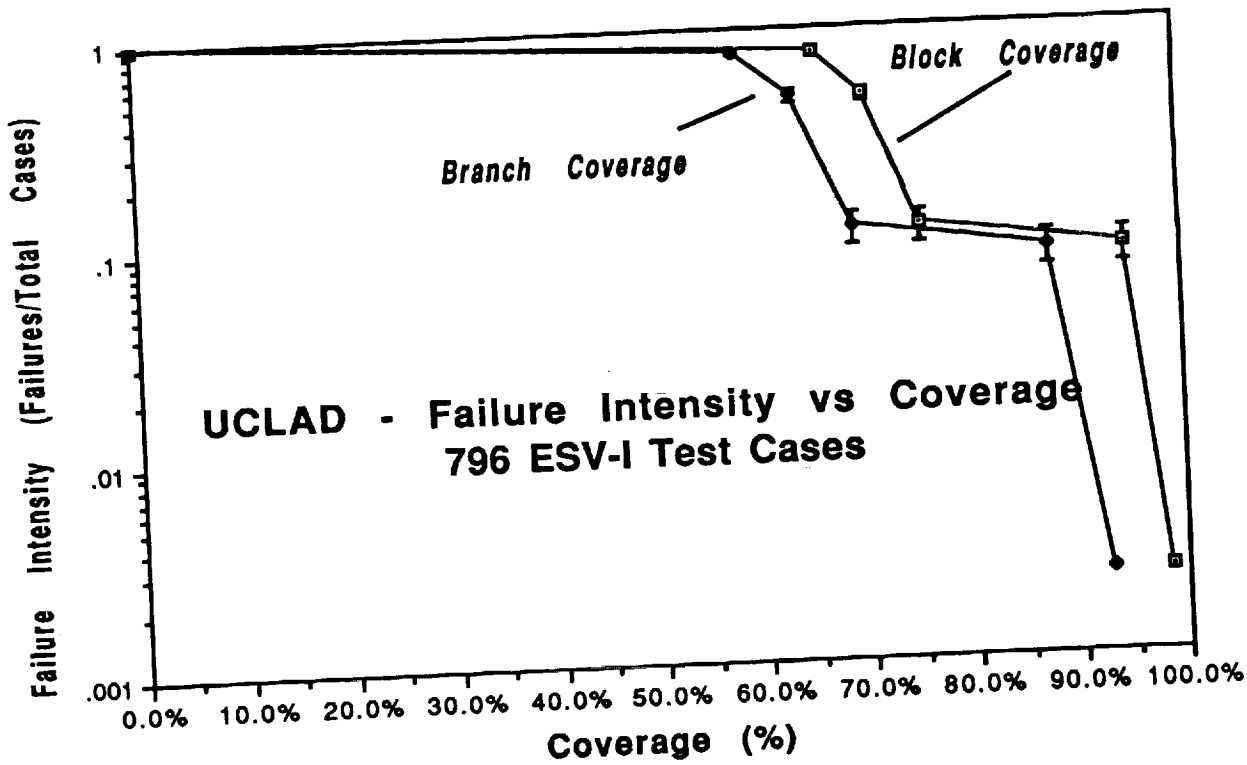
Coverage observed during random testing of a program from the 6-version set.


100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
Coverage
Blocks
Branches
DU-pairs & C-uses
p-use
DUD-chains
NCSUC - 1196 Certification Testing Cases
1
10
100
1000
10000
Test Cases

Work in Progress and Future Work

Coverage Based Reliability Models

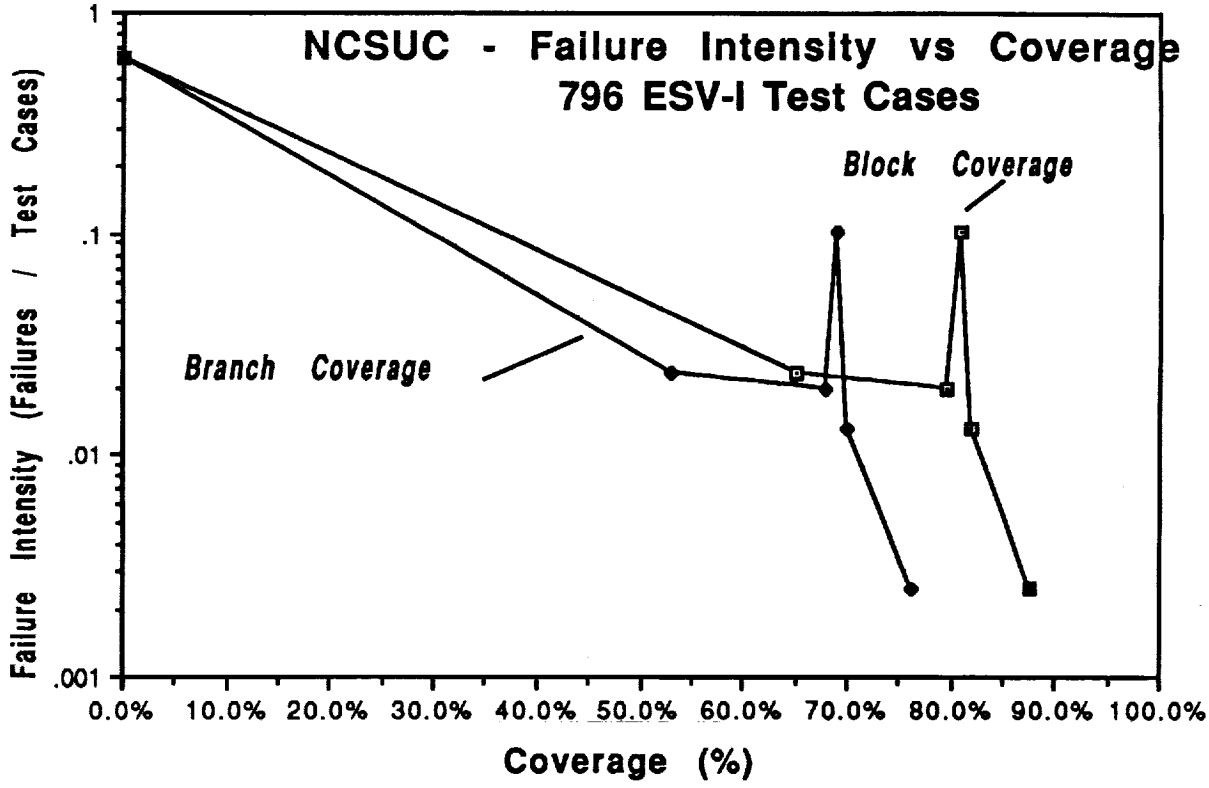
- Failure Intensity vs. Covered (tested) program functions or sub-functions.



- Function based reliability models.
- How to measure the extent of function synthesis provided by test cases?

- **One possibility is a combination of the code structure based metrics (control flow, data flow, hybrid) with testing time exposure to capture both the test quality and the reliability growth.**

- **Work on formulation of a two component (coverage, time) reliability models is in progress.**



Appendix II:
Extensions to the BGG Testing Coverage Tool

Abstract

BGG, Basic Graph Generation and Analysis tool, was originally developed to help studies of static and dynamic software complexity, and testing coverage metrics. It is composed of several stand-alone modules, it runs in UNIX environment, and currently handles static and dynamic analysis of control and data flow graphs (global, intra-, and inter-procedural data flow) for programs written in full Pascal. We describe additions to the structure of BGG in the form of the facility to generate and analyze reduced data-flow graphs, and the facility to perform Boolean and Relational Operator (BRO) testing analyis. Condition based testing focuses the testing process on predicates in a program. A new condition testing strategy called BRO testing was developed by Tai. To guide the test generation for BRO testing, an algorithm developed by Tai was incorporated into BGG. We are in the process of using the tool to collect data on the relationship between the software errors discovered in RSDIMU software and the BRO coverage achieved during RSDIMU testing.

I. Introduction

The system, BGG (Basic Graph Generation and Analysis system), was built as a research tool to help understand, study, and evaluate the many software complexity and testing metrics that have been proposed as aids in producing better quality software in an economical way [Appendix A]. BGG allows comparison of coverage metrics and evaluation of complexity metrics. It can also serve as a support tool for planning of testing strategies (e.g. stopping criteria), as well as for active monitoring of the testing process and its quality in terms of the coverage provided by the test cases used. It has now been extended with data-flow graphing analysis capabilities and with new structure based metrics for Boolean and Relational Operator (BRO) testing [Tai90] capabilities. Both will be used in the research on software desing testability. Section II of the report provides an overview of the data-flow additions to BGG. Section III gives a brief overview fo the BRO theory and some details concerning the implementation of the metric in the BGG context. Section IV illustrates some of the tool capabilities through examples. A brief description of the tool and its current "man" pages are shown in Appendix I of this report.

II. Data-flow extensions

Originally, BGG was written for control flow graph analyses with data flow computations based on the control-flow graph information for all variables in a given program. In a control flow graph the nodes represent the basic blocks and the arcs represent possible threads of execution, where a

basic block is defined to be a series of statements without branching. Each conditional statement is considered its own basic block that has multiple target blocks which are determined at runtime.

The methodology used by BGC is to first build control flow graphs for each of the procedures of a given program. Each node of the graph indicates which variables were assigned (defined) a value or were used in the calculation of another variable. From these graphs, the data flow is calculated. The information calcuated includes definition-use (du) pairs, definition-use-defintion (dud's) triplets, p-uses, and c-uses.

The problem with this methodology was that for most variables not all nodes or paths of the control flow graph are relative. That is, for a given variable, entire sections of the graph would not contain either a definition or use of that variable, however, the algorithm used in bgc required that all possible paths be explored. This exhaustive search is necessary but very time consuming.

To help reduce the time required to analyze a program, the control flow graph for each variable is first reduced to a data flow graph before analysis is performed. A data flow graph is one where only the nodes that contain either a definition or use a variable remain. This transformation, therefore, removes all unnecessary nodes and arcs from the graph, reducing the overall size.

This facility is available as a separate option on invocation of BGG (see Appendices A and Appendix B of this paper).

III. BRO Extensions

Boolean and Relation Operator testing focuses on the detection of Boolean and relational operator errors in a condition. An algorithm for selection of a minimum set ot testing constraints is described in [Tai90]. A minimum set of testing conditions consists of two sets of tests, min_t, or the smallest adequate set that makes the expression true, and min_f, or the smallest set that makes the expression false. BRO algorithms has been incorporated in a special version of BGG. Both the algorithms and the metric are still under study. The following example illustrates the algorithm.

The following describes the steps in the calculation of a BRO set for the expression "(x < 0) and (y > 0) or (x < y)". Braces { } denote a set, and each element of the set is a relational operator. The true and false sets calculated by our program written to automate the process of computation onf min_t and min_f sets are:

```
min_t =
{
   { <,>,> },
   { <,<,< }
}

min_f =
{
   { <,=,> },
   { <,<,> },
   { =,>,> },
   { >,>,= }
}
```

The **BRO** set is the combination of the min_t and min_f sets. Note that min_t are the test cases that make the expression true and the min_f make it false. This set is only one of many possible sets that could be formed for the same expression. The following description clearifies how the numerous sets could be formed.

The first step in calculating the BRO set is to form the min_t and min_f for the various subexpression. A bottom-up parse is used to ensure that proper presidence and grouping is maintained. The min_t and min_f for a minimal subexpression are:

```
x < y     --   min_t = { (<) }          min_f = { (=),{>) }
x > y     --   min_t = { (>) }          min_f = { (=),{<) }
x = y     --   min_t = { (=) }          min_f = { (<),{>) }
x <> y    --   min_t = { (<),(>) }      min_f = { (=) }
x         --   min_t = { (T) }          min_f = { (F) }
```

For the expression mentioned above, the algorithm starts by generating the min_t and min_f for the first subexpression , "(x < 0)", and places them on two separate stacks, one for each of the minimal sets. As the expression is read, the boolean operator "and" is remembered for processing after the min sets for the second subexpression have been determined and placed on the appropriate stacks.

The stacks would appear as follows:

```
min_t(y > 0) = { (>) }
min_t(x < 0) = { (<) }
-----------------------
   true stack

min_f(y > 0) = { (=),(<) }   (tos)
min_f(x < 0) = { (=),(>) }
-----------------------
        false stack
```

The next is to combine the two entries on the stack to produce a single set which will represent the subexpression "(x < 0) and (y > 0)". To produce the min_t of this "and" subexpression, the two

min_t sets are combined using the onto function described in [1]. The resulting set is constructed such that each element of both sets must appear at least once in the final set. For our example, the combination is trivial in that the resulting set is { (<,>) }. This set will make the sub expression true.

Calculating the min_f is a little trickier. From the min_t of the combined subexpression, choose two different members unless there is only one as in our case. Replace the left most operator with a member of the left subexpression's min_f(x < 0) to make the entire subexpression false. Repeat for all elements in the min_f. With the second element choosen from the resulting min_t, replace the second operator with the elements of the right most's min_f. The resulting stacks are:

```
min_t((x < 0) and (y > 0) = { (<,>) }
-------------------------------------
            true stack

min_f((x < 0) and (y > 0) = { (=,>), (>,>),
                             (<,=), (<,<) }
-------------------------------------
            false stack
```

What is actually happening here is that from the element(s) chosen from the min_t, we are holding one of the subexpression true and forcing the other to be false by using the min_f sets from the two subexpression. The next step is to hold the other subexpression true while making the other one false. From this algorithm, there is no way that a set element of (=,=) could be generated for any BRO set.

Once calculated, the next boolean operator is observed and stored for processing once another subexpression has been completed. In our example, the subexpresion (x < y) is encountered and the appropriate min sets are placed on the stack. Now the processing of the "OR" boolean operator is just opposite of the "AND" in that the two min_f sets are combined using the onto function and the elements of the two min_t currently on the stack are combined with two different elements from the resulting min_f.

So, before processing the stacks are :

```
min_t(x < y)                = { (<) }
min_t((x < 0) and (y > 0) = { (<,>) }
-------------------------------------
            true stack
```

```
min_f(x < y)                          = { (=), (>) }
min_f((x < 0) and (y > 0)) = { (>,>), (=,>),
                                        (<,<), (<,=) }
-----------------------------------
              false stack
```

The *onto* funtion states that each element of both sets must appear at least one time in the resulting set. Our strategy was based on linked list. Each element in a the set was a node in the linked list. The first element of each list were combined, the second element, and so forth until one list was exhausted. When this happened the last element of that list is combined with the remaining elements from the other list. The resulting min_f = { (>,>,=), (=,>,>), (<,<,>), (<,=,>) }.

Now, to calculate the min_t, two independent elements of the min_f set are selected. The algorithm used selects (<,<,>) and (<,=,>) but the choices are arbitray. The three members of these two elements represent the conditions which will make the entire expression false. Now, the theory replaces one of the subexpressions with a condition which will make that subexpression true, resulting in the entire expression being true (since we are or'ing the subexpressions). The left subexpression ((x < 0) and (y > 0)) can be made true by replacing the first two members of either but not both elements with the min_t for that subexpresion, which is (<,>). The resulting element is (<,>,>).

The element not chosen above is combined with the min_t of the subexpression (x < y) with the rightmost member being replaced. This will produce an element (<,<,<). As mentioned, the union of the min_t and min_f are the BRO set for this expression. Although this is a BRO set, the selection algorithm can produced different BRO set. Even the ordering of the element members can produce different BRO set.

To date, we have implemented the production of a single BRO set. After statically calculating this set, the code is instrumented in such a manner that at runtime the BRO set elements can be match if they are executed. The implemention was interesting for a couple of reasons. First, the theory is type independent but the implementation was not able to be. That is, in pascal, variables of pointer or boolean type may be compared to determine equality/ inequality, but not greater or less than. This caused problem since variable types are not considered by the tool and the selection of the min_t and min_f set for the inequality operator does not then hold for variables of this type.

The other problem had to do with handling the various language construct. The theory does not provide min sets for the "for loop" or case statement. In these cases, we chose to handle the for

loop as the expression x <= upper bound or x >= lower bound depending if x in increment or decremented. For the case statement, we chose to handle it as a series of x = value with the each case statement having a new block number. This criteria produces only branch coverage but no better.

Our future work will be to extend the current tool to calculate all possible BRO set for a given expression. Once calculated and statistics generated, a comparison of the coverage provided by the BRO sets will be performed. We are interested in determining if a particular bro set "out performs" another in providing better code coverage and is there a selection criteria that can be generalized. A second issue we wish to examine is the inequality operator and its relationship to the bro set. Here, we wish to use the min sets of min_t = { (<>) } and min_f = { (=) } and compare the results to the current usage. If we find that the min sets must be as described by [Tai90], then type information will need to be used in the implementation of BRO.

[Tai90] K.C. Tai, "Theory of Condition-Based Software Testing", NCSU Computer Science Technical Report, TR-90-11 (September 91 revision)

Section Appendix A

BGG: A Testing Coverage Tool[1]

Mladen A. Vouk, Robert. E. Coyle[2], Dana Borger

North Carolina State University
Department of Computer Science, Box 8206
Raleigh, N.C. 27695-8206

Extended Abstract

The system, BGG (Basic Graph Generation and Analysis system), was built as a research and teaching tool to help understand, study, and evaluate the many software complexity and testing metrics that have been proposed as aids in producing better quality software in an economical way. BGG allows comparison of coverage metrics and evaluation of complexity metrics. It can also serve as a support tool for planning of testing strategies (e.g. stopping criteria), as well as for active monitoring of the testing process and its quality in terms of the coverage provided by the test cases used.

A simplified top level diagram of BGG is shown in Figure 1. BGG is composed of several modules which can be used as an integrated system, or individually given appropriate inputs, to perform static and dynamic analyses of local and global control and data flow in programs written in Pascal. The UNIX version of the tool currently handles full Berkeley Pascal, while the PC version accepts Turbo Pascal. Extension to C is planned. UNIX version of BGG is itself written in Pascal, C and UNIX C-shell script, while PC version is written in Turbo Pascal.

BGG pre-processor provides the user interface when the tool is used as an integrated system. It also performs some housekeeping chores (checks for file existence, initializes appropriate language tables and files, etc.), and prepares the code for processing by formatting it and stripping it of comments. The language tables are generated for the system once, during the system installation, and then stored. The front-end parsing is handled through the FMQ generator [Fis88]. This facility also allows for relatively simple customization of the system regarding different programming languages and language features. Also, each of the BGG modules has a set of parameters which

[1]Research supported in part by NASA Grant No. NAG-1-983
[2]Teletec Corporation, Raleigh, N.C.

can be adjusted to allow analyses of problems which may exceed the default values for the number of nodes, identifier lengths, nesting depth, table sizes, etc.

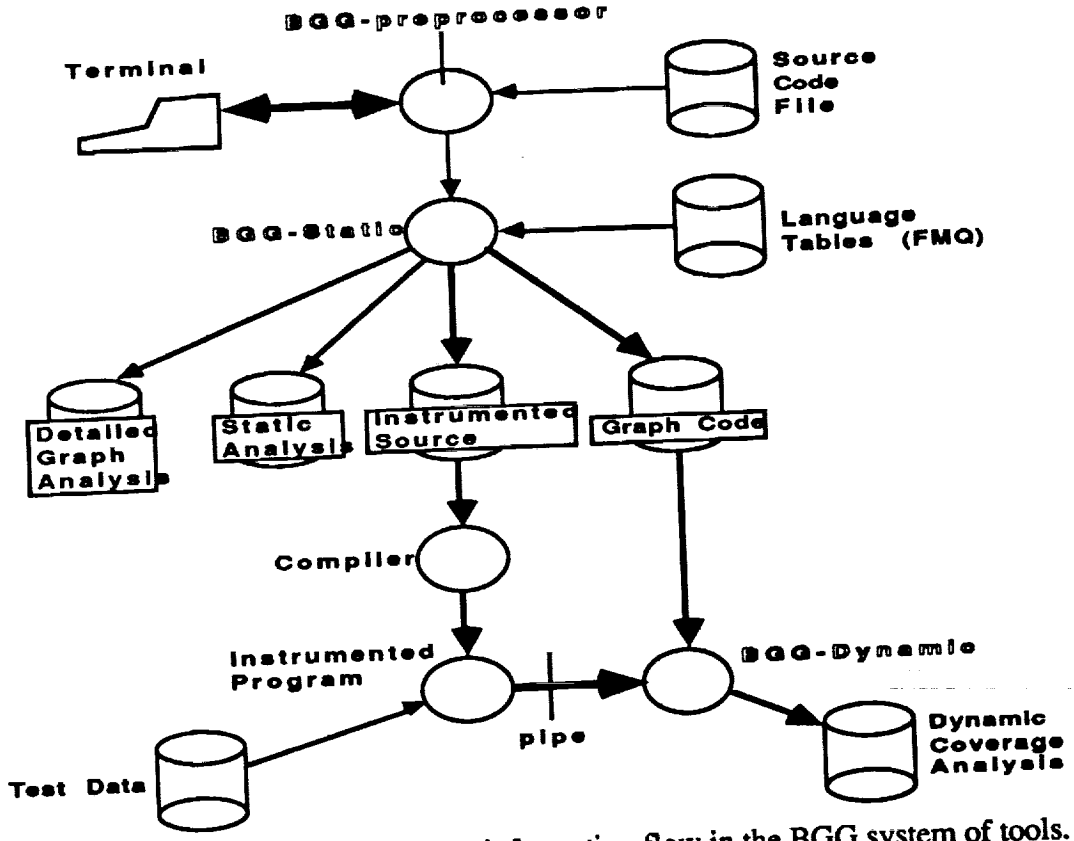


Figure 1. Schematic diagram of the information flow in the BGG system of tools.

Pre-processed code, various control information and language tables are used as input to the BGG-Static processor. This processor constructs control and data-flow graphs, and performs static analysis of the code. These graphs are the basis for all further analyses. Statistics on various metrics and control-flow and data-flow anomalies, such as variables that are used but never defined etc, are reported. BGG-Static also instruments the code for dynamic execution tracing.

When requested, BGG executes the instrumented code with provided test cases and analyzes its dynamic execution trace through BGG-Dynamic. The dynamic analysis output contains information (by procedures and variables) about the coverage that the test cases provide under different metrics.

When instrumenting code BGG inserts a call to a special BGG procedure at the beginning of each linear code block. It also adds empty blocks to act as collection points for branches. The

instrumentation overhead in executable statements is roughly proportional to the number of linear blocks present in the code. In our experience this can add between 50% and 80% to the number of executable lines of code. The run-time tracing overhead for the instrumented programs is proportional to the number of linear blocks of code times the cost of the call to the BGG tracing procedure. The latter simply outputs information about the block and the procedure being executed.

The raw run-time tracing information may be stored in temporary files, and processed by BGG-Dynamic later. However, often the amount of raw tracing information is so large that that it becomes impractical to store it. BGG-Dynamic can then accept input via a pipe and process it on-the-fly. Because BGG-Dynamic analyses may be very I/O, memory and CPU intensive, particularly in the case of data-flow metrics, interactive testing may be a slow process. Part of the problem lies in the fact that BGG is still a research tool and was not optimized. We expect that the next version of BGG will be much faster and more conservative in its use of memory. It will permit splicing of information from several short independent runs, so that progressive coverage can be computed without regression runs on already executed data.

Currently BGG computes the following static measures: counts of local and global symbols, lines of code (with and without comments), total lines in executable control graph nodes, linear blocks of code, control graph edges and graph nodes, branches, decision points, paths (the maximum number of iterations through loops can be set by the user), cyclomatic number, definition and use counts for each variable, definition-use (du) pair counts, definition-use-redefinition (dud) chain counts, count of definition-use paths, average definition-use path lengths, p-uses, c-uses, and all-uses. Dynamic coverage is computed for definition-use pairs, definition-use-redefinition chains, p-uses, c-uses and all-uses [Fra88]. Definition-use path coverage and path coverage for paths that iterate loops k times (where k can be set by user) will be implemented [Nta88]. There are several system switches which allow selective display and printing of the results of the analyses.

Outputs like the one shown in Figure 2 provide summary profile of each local and global symbol found in the code. How many times it was defined (or pseudo-defined), used (or pseudo-used), how many du-pairs it forms, how many d(u)d chains, etc. This static information can be used to judge the complexity of procedures, or the complexity of the use of individual variables. In turn, this information may help in deciding which of the variables and procedures need additional attention on the part of the programmers and testers.

BGG provides coverage information on the program level, and on the procedure level. Figure 3 illustrates output from the dynamic coverage processor "BGG-Dynamic", delivered in the

"Dynamic Coverage Analysis" output file, for a function called FPTRUNC and a set of 103 test cases. In the example some of the output information normally delivered by BGG has been turned off, e.g. all-uses.

For each procedure BGG-Dynamic first outputs a summary of branch coverage information: the block number, statement numbers encompassed by the block, the number of times the block was executed, and the execution paths taken from the block (node). For example, node 5 in Figure 7 was executed 724 times, 6 times to node 3, and 721 times to node 7. Branches which have not been executed show up as having zero executions.

The branch table is followed by a summary of coverage by metrics: coverage for non-empty blocks (blocks that have not been inserted by BGG), lines of code within executable nodes, and branch coverage. This is followed by coverage for data flow metrics by symbol name. The static definition, use, du-pair, d(u)d, p-use, etc. counts for a variable are printed along with the information on its the dynamic coverage expressed as percentage of the executed static constructs. For each identifier, this is followed by a detailed list and description of constructs that have not been executed (e.g. du-pairs or p-uses). Execution coverage output tables can be printed in different formats (e.g. counts of executed constructs, rather than percentages), and with different content (e.g. all-uses).

Apart from providing static information on the code complexity, and dynamic information on the quality of test data in terms of a particular metric, BGG can also be used to determine the point of diminishing returns for a given data set though coverage growth curves, and help in making the decisions on when to switch to another testing profile or strategy. We are currently using BGG in an attempt to formulate coverage based software reliability models by relating code complexity, testing quality (expressed through coverage), and the number of faults that have been discovered in the code. BGG is also an excellent teaching tool, and is used in the Software Engineering and the Software Testing and Reliability courses taught at North Carolina State University, Department of Computer Science.

[Fis88] C.N. Fisher and R.J. LeBlanc, *Crafting a compiler*, The Benjamin/Cummings Co., 1988.

[Fra88] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," IEEE Trans. Soft. Eng., Vol. 14 (10), pp 1483-1498, 1988.

[Nta88] S.C. Ntafos, "A Comparison of Some Structural Testing Strategies", IEEE Trans. Soft. Eng., Vol. SE-14 (6), pp 868-874, 1988.

Procedure: FPTRUNC

Local Symbol Name	Def Ct	Use Ct	DU Ct	DUD Ct	Cyclic DUD Ct	Acyclic DUD Ct	DU Len	DUD Len	Cyc Len	Acyc Len	Subpath Ct	Puse Ct	Cuse Ct
FPTRUNC	3	1	3	0	0	0	1.00	0.00	0.00	0.00	303		
X	1	3	3	0	0	0	1.88	0.00	0.00	0.00	13	2	2
LARGEVALUE	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	2	0
REMAINDER	3	9	14	9	2	7	4.14	5.33	4.50	5.57	12	12	8
POWER	3	6	16	6	3	3	2.61	2.83	3.00	2.67	16	10	11
BIGPART	2	2	4	4	2	2	1.00	1.00	1.00	1.00	8	0	4
TERM	1	2	2	2	2	0	1.50	2.00	2.00	0.00	4	0	2
Global Symbol Name													
ABS	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	0	1
MAXINT	1	1	1	0	0	0	1.67	0.00	0.00	0.00	13	2	0
TRUNC	1	2	2	0	0	0	1.40	0.00	0.00	0.00	13	0	2
Procedure Summary:	17	28	47	21	9	12	2.62	3.48	2.67	4.08	108	28	33

Total local symbols: 7
Total global symbols: 3
Total lines (in executable nodes): 27
Total lines (in non-empty xqt nodes): 25
Total blocks: 18
Total non-empty blocks: 16
Total edges: 22
Total branches: 10
Total decisions: 5
Total paths: 13
Cyclomatic number: 6

Figure 2. Sample static analysis

Proc. #: 8 Proc/Func Name: FPTRUNC

Block	Statements	Executions	Branches(Executions)
1	168 - 169	721	2(721)
2	170 - 170	721	3(0) 4(721)
3	170 - 171	0	18(0)
4	172 - 173	721	5(721)
5	174 - 174	724	6(3) 7(721)
6	174 - 175	3	5(3)
7	176 - 177	721	8(721)
8	178 - 178	721	9(0) 13(721)
9	178 - 179	0	10(0)
10	180 - 180	0	11(0) 12(0)
11	180 - 181	0	10(0)
12	182 - 185	0	8(0)
13	186 - 188	721	14(721)
14	189 - 189	721	15(66) 16(655)
15	189 - 190	66	17(66)
16	191 - 192	655	17(655)
17	193 - 193	721	18(721)
18	194 - 194	721	

Non-empty blks exec/total non-empty blks = 12 / 16 = 75.00%
Lines in executed nodes/total lines in executable nodes = 18 / 25 = 72.00%
Branches exec/total branches = 6 / 10 = 60.00%
Legend: Ct - Total static count, Def - Definitions, Use - Uses, DU - Definition-Use Pairs, DUD - Definition-Use-(Re)Definition Chains
PecentX - percentage of the static count executed with the current set of test cases, Puse - Predicate-Uses, Cuse - Computational Uses

Local Symbol Name	Def Ct	Use Ct	DU Ct	PercentX	DUD Ct	PercentX	Puse Ct	PercentX	Cuse Ct	PercentX
FPTRUNC	3	1	3	66.67	0	0.00	0	0.00	3	66.67
	DU pairs not executed:									
	Defined: 171		Used: 194							
X	1	3	3	66.67	0	0.00	2	100.00	2	50.00
	DU pairs not executed:									
	Defined: 168	Used: 171								
LARGEVALUE	1	1	1	100.00	0	0.00	2	50.00	0	0.00
	P-uses not executed:									
	Defined: 168		Used: 170		Successor: 3					
REMAINDER	3	9	14	42.86	9	22.22	12	33.33	8	37.50
	DU pairs not executed:									
	Defined: 183	Used: 183								
	Defined: 183	Used: 182								

Figure 3.Part of a dynamic analysis output.

Appendix B

VAX bgg(se)

NAME

bgg - performs static and dynamic analysis (execution coverage) of Pascal code.

SYNTAX

bgg file_name.p [up to two options]

DESCRIPTION

The bgg command compiles and performs static and dynamic analysis of Pascal code using a number of metrics.

This version of bgg is generated for analysis of Berkeley Pascal or its subset.

Analysis can be performed using program control graph or data-flow graphs for individual variables. Analysis is performed for global, inter- and intra-procedural control and data flow. A summary is also provided for the whole program.

Some of the active static metrics are: statement, line, & comment counts, cyclomatic number, branch, definition-use pair and path count.

Some of the active dynamic coverage metrics are: statement, branch definition-use pair, p-uses, and c-uses coverage.

To get more help on execution options type: bgg -h

More complete documentation is available in the form of users manual and a paper describing the tool.

It is possible to customize the bgg driver to access some analysis options which are not available in the default mode.

OPTIONS

There are four processors that can be controlled: bgg-shell, bgg-bgc the graph generator, bgg-static the static analyzer, and bgg-dynamic the dynamic analyzer.

** bgg-shell options:

bgg-shell runs bgc, bgg-static and bgg-dynamic in default modes unless otherwise is specified through options.

default option: graph generation only
* file_name.p file must be available
other options: -x generate graph, static, and dynamic analysis
* file_name.p file must be available

```
                    -s  static analysis only
                        * all graph files must be available
                    -d  dynamic analysis only
                        * file_name.p.probe file must be available
                        * all graph files must be available
                    -a  static and dynamic analysis only
                        * file_name.p.probe file must be available
                        * all graph files must be available
                    -r  both static and dynamic analysis is
                        performed on reduced (data-flow) graphs
                    -h  help (this screen)
    warning: filenames "tables", "llgenout", "bgctables" are reserved names
             and will be deleted if they exist in the current directory

  ** bgg-bgc options:

     usage: bgc [options]
     required file:  tables, llgenout  test.p or option -f
     default:   all -pxxx options are on.
     options:   -v      print version and stop
                -prdef  turn off analysis of predefined functions
                -ppdef  turn off analysis of parameter pseudo-definitions
                -ppuse  turn off analysis of parameter pseudo-uses
                -pgdef  turn off analysis of global pseudo-definitions
                -pguse  turn off analysis of global pseudo-uses
                -pcdef  turn off analysis of constant pseudo-definitions
                -f      fname.p, where fname.p is source code
                -h      help (this screen)

  ** bgg-static options:

     usage: dustatic [options] < bgctables
     required file:  bgctables as standard input
     default:   control-flow analysis, iteration depth is one
     options:    -r    for analysis use reduced graph; default: full control
graph
                 -v  print program version only
                 -i  xx
                     set depth of loop iteration to xx; default is 1
                 -p  fname
                     fname contains list of procedures or functions
                     by their ordinal number, one a line, which are NOT
                     to be processed during static analysis;
                     default is to process all procedures/functions
                 -h  help (this screen

  ** bgg-dynamic options:

     usage: dudynamic [options]
     required files: probe, bgctables
     default:   control-flow analysis
     options:   -r  for analysis use reduced graph
                -v  print program version only
                -h  help (this screen
```

BUGS

This is a field testing release of bgg. Please remember that bgg is a research and teaching tool still under development. It contains some bugs we know about, and probably many we do not know about. So exercise care and check the results for consistency and sense.

Please report all anomalies to

vouk@adm.csc.ncsu.edu

bgg will only take complete programs which do not take input directly from the keyboard and output drectly to the screeen. All I/O has to be indirect (via files).

bgg programs must have the (input,output) part.

NOTE

Under VAX Ultrix bgg is very slow, so be patient. To start with, analyze only very small code. Code to be analyzed must be a complete program.

Appendix III: An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the Presence of Failure Correlation*

(submitted to the Special Issue of Journal of Computer and Software Engineering, March 1992)

Mladen A. Vouk[3],David F. McAllister[1], David E. Eckhardt[4], Kalhee Kim[1]

Abstract

Reliability of fault-tolerant software system implementations based on Consensus Voting and Consensus Recovery Block strategies is evaluated using a set of independently developed functionally equivalent versions of an avionics application. The strategies are studied under conditions of high inter-version failure correlation and with program versions of medium-to-high reliability. Comparisons are made with classical N-Version Programming that uses Majority Voting, and with Recovery Block strategies. The empirical behavior of the three schemes is found to be in good agreement with theoretical analyses and expectations. Consensus Voting and Consensus Recovery Block based systems are found to perform better and more uniformly than corresponding traditional strategies, i.e. Recovery Block and N-Version Programming that uses Majority Voting. This is the first experimental evaluation of the system reliability provided by Consensus Voting, and the first experimental study of the reliability of Consensus Recovery Block systems composed of more than three versions.

Key Words: Consensus Recovery Block, Consensus Voting, System Reliability, Software Fault-Tolerance, Correlated Failures

1. Introduction

Redundancy can be used to provide fault-tolerance in software systems. Several independently developed but functionally equivalent software versions are combined in various ways in an attempt to increase system reliability. Over the years simple majority voting and recovery block based software fault-tolerance has been investigated by a number of researchers, both theoretically [e.g., Ran75, Avi77, Grn80, Eck85, Sco87, Deb88, Lit90] and experimentally [e.g., Sco84a,84b, Bis86, Kni86, Shi88, Eck91]. However, studies of more advanced models such as Consensus Recovery Block [e.g. Sco84a, Sco87, Deb88, Bel90], Community Error Recovery [e.g. Tso86, 87, Nic90], Consensus Voting [McA90] or Acceptance Voting [Ath89, Bel90] are less frequent and mostly theoretical in nature. One of the principal concerns with all redundancy based software fault-tolerance

*Research supported in part by NASA Grant No. NAG-1-983

[3]Department of Computer Science, North Carolina State University, Box 8206, Raleigh, NC 27695-8206
[4]NASA Langley Research Center, MS 478, Hampton, VA 23665

strategies is their performance in the presence of failure correlation among the versions comprising the system.

In a recent study Eckhardt et al. [Eck91] addressed the issue of reliability gains offered through classical majority-based N-Version Programming using high reliability versions of an avionics application under conditions of small-to-moderate inter-version failure dependence. In this paper we discuss system reliability performance offered by more advanced fault-tolerance mechanisms under more severe conditions. The primary goal of the present work is mutual comparison of different experimental implementations of Consensus Recovery Block in the presence of inter-version failure correlation, and a comparison of Consensus Voting and Consensus Recovery Block with more traditional schemes such as N-Version Programming with Majority Voting. We report on the relative reliability performance of Consensus Voting and Consensus Recovery Block in an environment where theoretically expected effects could be easily observed, i.e., under the conditions of strong inter-version failure coupling using medium-to-high reliability software versions of the same avionics application as was employed in [Eck91]. To the best of our knowledge this study is the first experimental evaluation of the Consensus Voting techniques, and the first experimental study of the reliability of Consensus Recovery Block systems composed of more than three versions

In section 1 of the paper we provide an overview of software fault-tolerance techniques of interest, in section 2 we discuss different voting approaches and the question of correlated failures, in section 3 we describe the experimental environment and present the results. Summary and conclusions are given in section 4.

1.1 Recovery Block and N-Version Programming

One of the earliest fault-tolerant software schemes is **Recovery Block** [e.g., Ran75, Deb86]. In Recover Block independently developed functionally equivalent versions are executed in sequence and the output is passed to an acceptance test. If the output of the first version fails the acceptance test, then the second, or first backup, software version is executed and its output is checked by the acceptance test, etc. In the case where the outputs of all versions are rejected the system fails. One problem with this strategy is the sequential nature of the execution of versions. This was recently addressed by [Bel91]. Another is finding a simple and highly reliable acceptance test which does not involve the development of an additional software version. Another basic fault-tolerant software strategy, **N-version Programming** [e.g., Avi77, Avi85], proposes parallel execution of independently developed functionally equivalent versions with adjudication of their outputs by a voter. One problem with all strategies based on voting is that situations can arise where there is an

insufficient number of agreeing versions and voting fails simply because the voter cannot make a decision.

1.2 Consensus Recovery Block

Scot et al. [Sco87] developed a hybrid software fault-tolerance model called **Consensus Recovery Block.** The system executes independently developed functionally equivalent versions on the same input in series or parallel. Then it attempts to vote on the returned results. If the voting module cannot make a decision, the system reverts to Recovery Block. The strategy is depicted in Figure 1, where the number of versions in the system is N, 1 - α is the probability that a version gives correct result, β_1 is the probability that acceptance test rejects a correct result, β_2 is the probability that acceptance test accepts an incorrect result. In general, Consensus Recovery Block offers system reliability superior to that provided by N-Version Programming [Sco87, Bel90]. However, Consensus Recovery Block, like N-Version Programming does not resolve the problem of a voter which returns a wrong answer because several versions produce identical-and-wrong answers or there is not a majority as might be the case when there are multiple correct outputs.

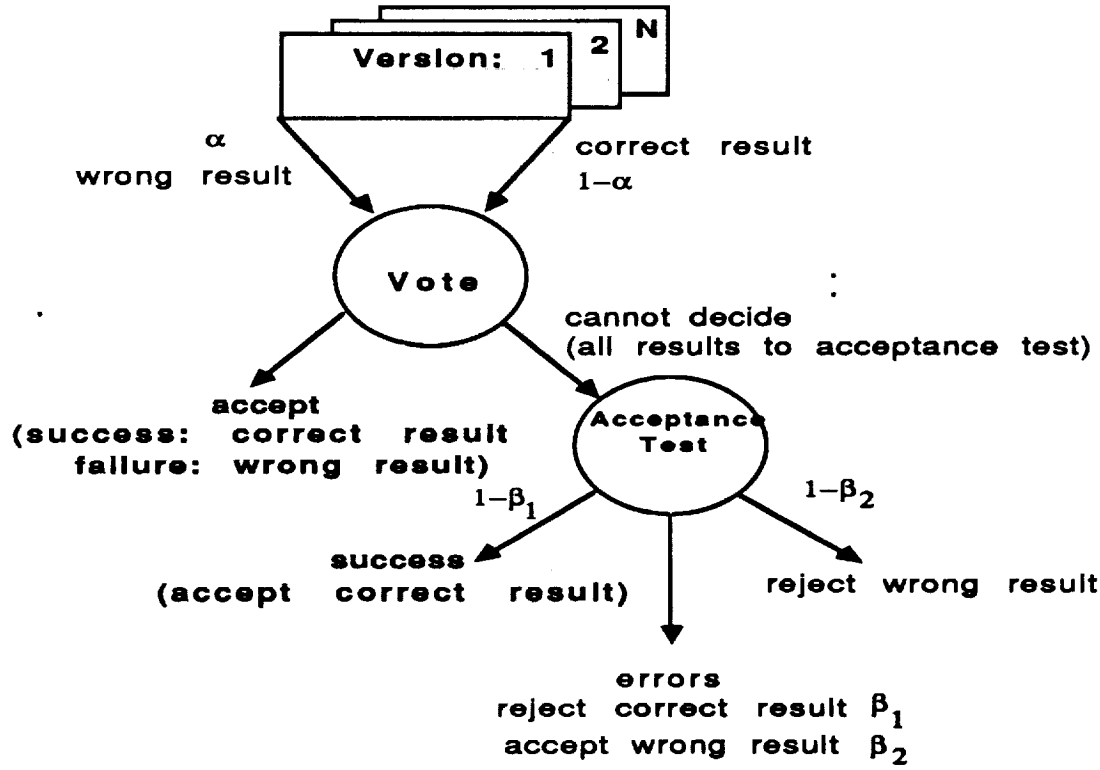


Figure 1. Consensus Recovery Block model.

2. Adjudication Strategies

2.1 Majority and 2-out-of-N Voting

In an m-out-of-N fault-tolerant software system the number of versions is N, and m is the agreement number, or the number of matching outputs which the adjudication algorithm (e.g. voting) requires for system success [e.g. Tri82, Eck85]. In the past N was rarely larger than 3, and m was traditionally chosen as $\frac{N+1}{2}$ for odd m. In general, in **Majority Voting**, $m = \lceil \frac{N+1}{2} \rceil$, where $\lceil\ \rceil$ denotes the ceiling function. Scott et al. [Sco87] showed that, if the output space is large, and true statistical independence of version failures can be assumed, there is no need to choose $m > 2$ regardless of the size of N, although larger m values offer additional benefits. We use the term **2-out-of-N Voting** for the case where agreement number is m=2. In this experiment we do not have statistical independence of version failures. Hence, this voting technique is used only when showing upperbounds for reliabilities of the systems. In a model based on software diversity and a voting strategy there is a difference between correctness and agreement. McAllister et al. [McA90] distinguish between agreement and correctness and develop and evaluate an adaptive voting strategy called **Consensus Voting**. This strategy is particularly effective in small output spaces because it automatically adjusts the voting to the changes in the effective output space cardinality. They show that for $m \geq 2$ the majority voting strategy provides lowerbound on the reliability provided by Consensus Voting, and 2-out-of-N upperbound .

2.2 Consensus Voting

The theory of Consensus Voting is given in [McA90]. In Consensus Voting the voter uses the following algorithm to select the "correct" answer:

- If there is a majority agreement ($m \geq \lceil \frac{N+1}{2} \rceil$, $N>1$) then this answer is chosen as the " correct" answer.
- Otherwise, if there is a unique maximum agreement, but this number of agreeing versions is less than $\lceil \frac{N+1}{2} \rceil$, then this answer is chosen as the "correct" one.
- Otherwise, if there is a tie in the maximum agreement number from several output groups then
  - if Consensus Voting is used in N-Version Programming one group is chosen at random and the answer associated with this group is chosen as the "correct" one.
  - else if Consensus Voting is used in Consensus Recovery Block all groups are subjected to an acceptance test which is then used to choose the "correct" output.

In [McA90] it is shown that the strategy is equivalent to Majority Voting when the output space cardinality is 2, and to 2-out-of-N voting when the output space cardinality tends to infinity provided

the agreement number is not less than 2. It is also proved that, in general, the boundary probability below which the system reliability begins to deteriorate as more versions are added is $\frac{1}{r}$, where r is the cardinality of the output space.

2.3 Coincident Failures and Inter-Version Failure Correlation

When two or more functionally equivalent software components fail on the *same* input case we say that a *coincident* failure has occurred. When two or more versions give the same incorrect response, to a given tolerance, we say that an *identical-and-wrong* answer was obtained. If measured probability of the coincident version failures is significantly different from what would be expected by chance, assuming failure independence model, then we say that the observed version failures are *correlated* or *dependent* [Tri82, Eck85, Lit90].

Experiments have shown that inter-version failure dependence among independently developed functionally equivalent versions may not be negligible in the context of current software development and testing strategies [e.g., Sco84a, Kni86, Eck91]. There are theoretical models of the classical majority based N-Version Programming model which incorporate inter-version failure dependence [e.g., Eck85, Lit90]. However, most of the theory for advanced software fault-tolerance strategies is derived under the assumption of inter-version failure independence, and failure independence of acceptance tests with respect to versions and each other (if more than one acceptance test is used). Still, the behavior of the strategies in the presence of failure correlation can be deduced from these simple models by extrapolation from their behavior in extreme situations. Therefore, it is interesting to see if the effects and special events that can be anticipated from analytical considerations can actually be observed in real multiversion software.

For example, in the case of implementations involving voting, presence of correlated failures produces an effect which is usually equivalent to either a reduction, or an increase in the average cardinality of the space in which voting takes place. An increased probability of coincident but different incorrect answers will tend to increase the average number of distinct responses offered to a voter for an input, while an increased probability of coincident identical-and-wrong failures will tend to decrease the voting space from what would be expected based on the cardinality of the application output space and version reliabilities (assuming versions are statistically independent). In a model based on failure independence the effects can be simulated, at least in part, through reduction or increase in the model output space size.

To see this consider the following. Assume that all individual version failure probabilities in an N-tuple are mutually independent [Tri82], have identical failure probability (1-p) over the usage (test) distribution, and have the same probability of occurrence of each program output failure state given by $\frac{1-p}{r-1}$, where: r is the size of the program output space, there is a unique success state j=1, and there are r-1 failure states, j=2,..,r. When r=2 (binary output space) all failures, and what is more important all coincident failures of the N-tuple versions, result in identical-and-wrong answers. On the other hand, under the above assumptions a large value of r translates into low probability that two incorrect answers are identical (in the analytical and simulation examples given later in this paper an "r = infinity" implies that the probability of obtaining identical-and-wrong answers is zero). This, in turn, implies higher probability that responses from coincidentally failing versions are different, and also increases the average size of the voting space when coincident failures occur. Of course, the voting space size is bounded by the number of versions in the N-tuple. An increase in the number of coincident version failures can be simulated, in part, by reduction in the value of p which shifts the peak of envelope of the independent coincident failures profile closer to N. However, in general, models based on the assumption of failure independence do not capture strong non-uniform failure coupling that can occur between two or more versions in practice (e.g. sharp spikes seen in the experimental trace in Figure 3) because the causes of the coupling are different (e.g. identical-and-wrong responses are the result of a fault rather than a basic change in the output space of the problem, although the effect may be the same).

An added dimension is failure correlation between an acceptance test and the N-tuple versions, or lack of mutual independence when two or more acceptance tests are used. The effects can, again only in part, be simulated by lowering reliability of the model acceptance test.

Nevertheless, we would expect that many of the effects observed in the experiments conducted in a high inter-version correlation environment would in the simple theoretical models correspond to small output space (r) and low p effects. Similarly, we would expect that implementations composed of versions that exhibit low mutual inter-version failure correlation would exhibit many characteristics that correspond to model computations based on large r values.

3. Empirical Results

In this section we discuss experimental data on reliability of N-Version Programming systems that use Consensus Voting (NVP-CV), and on Consensus Recovery Block systems that use either Majority Voting (CRB-MV), or Consensus Voting (CRB-CV). Consensus Voting and Consensus

Recovery Block are compared with N-Version Programming that uses Majority Voting (NVP-MV) and with Recovery Block (RB).

3.1 Experimental Environment

Experimental results are based on a pool of twenty independently developed functionally equivalent programs developed in a large-scale multiversion software experiment. We used the program versions in the state they were immediately after the unit development phase [Kel88], but before they underwent an independent validation (or acceptance) phase of the experiment [Eck91]. This was done to keep the failure probability of individual versions relatively high (and failures easier to observe), and to retain a considerable number of faults that exhibit mutual failure correlation in order to high-light correlation based effects. The nature of the faults found in the versions is discussed in [Vou90].

For the study we generated subsets of program N-tuples with: 1) similar average[5] N-tuple reliability, and 2) a range of average N-tuple reliabilities. We use the average N-tuple reliability to focus on the behavior of a particular N-tuple instead of the population (pool) from which it was drawn, and to indicate approximate reliability of corresponding mutually independent versions. In this paper we report on 3, 5 and 7 version fault-tolerant software systems. The subset selection process is described in Appendix I.

In conducting our experiments we considered a number of input profiles and different combinations of versions and output variables. Failure rate estimates based on the three most critical output variables (out of 63 monitored) are shown in Table 1. Two test suites each containing 500 uniform random input test cases were used in all estimates discussed in this paper. The sample size is sufficient for the version and N-tuple reliability ranges on which we report here. One suite, which we call Estimate-I, was used to estimates of individual version failure rates (probabilities), N-tuple reliabilities, select acceptance test versions, select sample N-tuple combinations, and compute expected "independent model" response. The other test suite, Estimate-II, was used to investigate the actual behavior of N-tuple systems based on different voting and fault-tolerance strategies. Recovery Block, and Consensus Recovery Block studies require an acceptance test. We used one of the developed versions as an acceptance test. This provided correlation not only among versions, but

---

[5]Average N-tuple reliability estimate is defined as $\bar{p} = \sum_{i=1}^{N} \frac{\hat{p}_i}{N}$, and the corresponding estimate of the standard deviation of the sample as $\hat{\sigma} = \sqrt{\sum_{i=1}^{N} \frac{(\bar{p}-\hat{p}_i)^2}{N-1}}$, where $\hat{p}_i = \sum_{j=1}^{k} \frac{s_i(j)}{k}$ is estimated reliability of version i over the test suite composed of k test cases, $s_i(j)$ is a score function equal to 1 when version succeeds and 0 when it fails on test case j, and $1-\hat{p}_i$ is the estimated version failure probability.

also between the acceptance test and the versions. Acceptance test versions were selected first, then N-tuples were drawn from the subpool of remaining versions. The fault-tolerant software algorithms of interest were invoked for each test case. The outcome was compared with the correct answer obtained from a "golden" program [Kel88, Vou90] and the frequency of successes and failures for each strategy was recorded.

Table 1. Version failure rates.

Version	Failure Rate*	
	Estimate I	Estimate II
1	0.58	0.59
2	0.07	0.07
3	0.13	0.11
4	0.07	0.06
5	0.11	0.10
6	0.63	0.64
7	0.07	0.06
8	0.35	0.36
9	0.40	0.39
10	0.004	0.000
11	0.09	0.10
12	0.58	0.59
13	0.12	0.12
14	0.37	0.38
15	0.58	0.59
16	0.58	0.59
17	0.10	0.09
18	0.004	0.006
19	0.58	0.59
20	0.34	0.33

(*) Based on the 3 most important output variables, "best.acceleration". Each column was obtained on the basis of a separate set of 500 random cases.

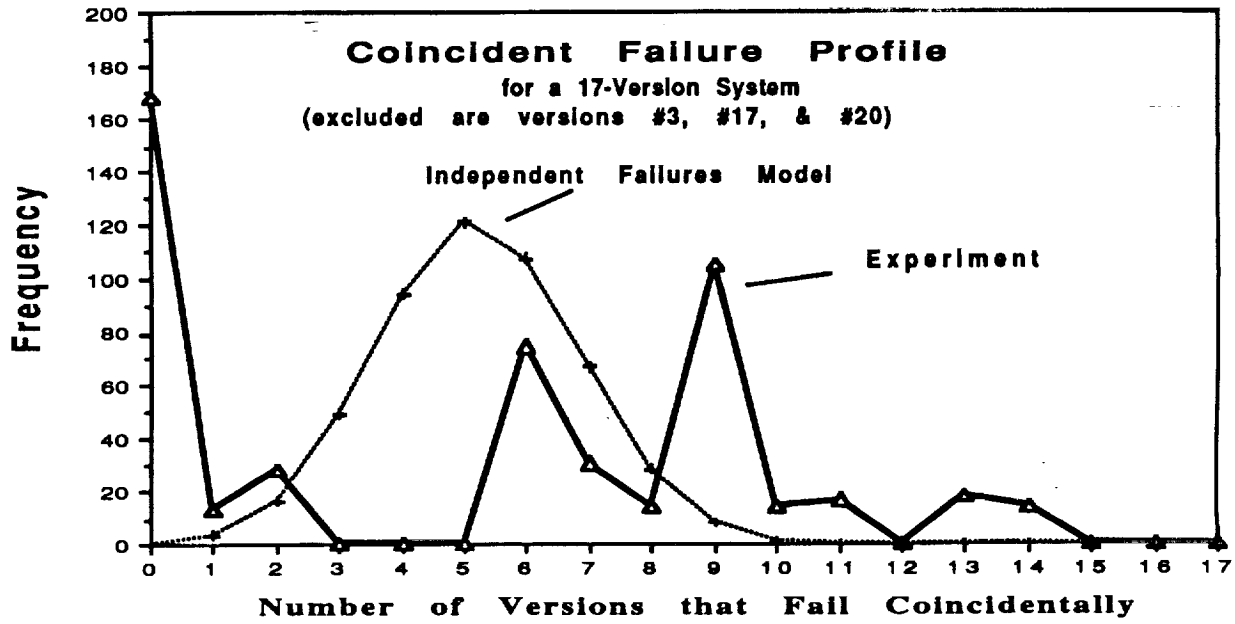


Figure 2. Example of a joint coincident failure profile.

Table 2. Frequency of empirical coincident identical-and-wrong (IAW) events over 500 test cases for the set of 17 versions shown in Figure 2. The span is the number of versions that coincidentally returned a IAW answer.

The Span of IAW Events																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Frequency																
2049	164	1	16	1	1	2	15	0	0	0	0	0	0	0	0	0

The failure correlation properties of the versions can be deduced from their joint coincident failure profiles, and the corresponding IAW response profiles. For example, Figure 2 shows the profile for a 17 version subset (three versions selected to act as acceptance tests are not in the set). The abscissa represents the number of versions that fail coincidentally, and the ordinate is the frequency of the event over the 500 samples. Also shown is the expected frequency for the model based on independent failures, or the "binomial" model [Tri82]. The deviation from the expected "independent" profile is obvious. For instance, we see that the frequency of the event where 9 versions fail coincidentally is expected to be about 10. In reality, we observed about 100 such events. Table 2 summarizes the corresponding empirical frequency of coincident IAW responses. For example, in 500 tries there were 15 events where 8 versions coincidentally returned an answer which was wrong yet identical within the tolerance used to compare the three most critical (real)

variables. Both, Figure 2 and Table 2, are strong indicators of a high degree of inter-version failure dependence in the version set we used.

3.2 Consensus Voting

Theory predicts that Consensus Voting is always either as reliable, or more reliable, than Majority Voting. In binary output space Consensus Voting reduces to Majority Voting and cannot improve on it. But for r > 2 Consensus Voting is expected to offer reliability higher than Majority Voting. Theory also predicts that in N-Version Programming systems composed of versions of considerably different reliabilities both Majority Voting and Consensus Voting would have difficulty providing reliability that exceeded that of the most reliable, or "best", component although Consensus Voting would still perform better than Majority Voting [McA90].

Figures 3 and 4 illustrate the observed relationship between N-Version Programming with Consensus Voting and Majority Voting system successes over a range of average N-tuple reliabilities for 3-version and 7-version systems respectively. The "ragged" look of the experimental traces is partly due to the small sample (500 test cases), but also due to the presence of very highly correlated failures. The experimental behavior is in good agreement with the trends indicated by the theoretical Consensus Voting model based on failure independence.

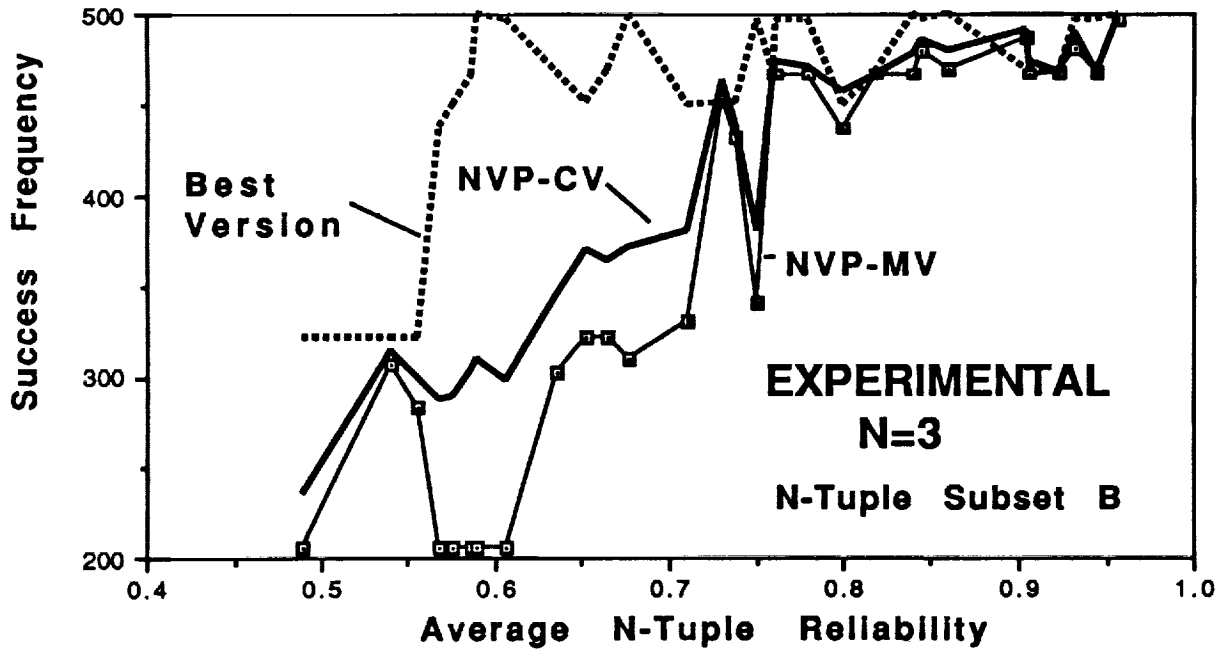


Figure 3. System reliability by voting (N=3).

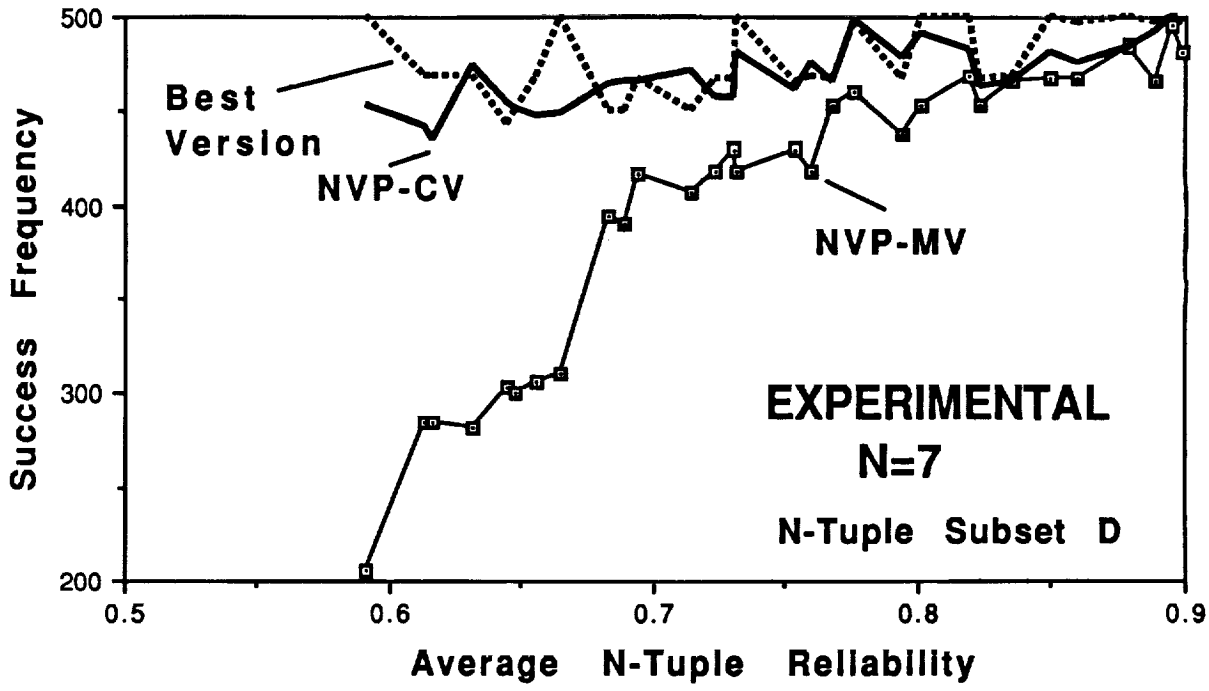


Figure 4. System reliability by voting (N=7).

For instance, we see that for N=3, N-Version Programming has difficulty competing with the "best" version when the average N-tuple reliability is low. Note that the "best" version was not pre-selected based on Estimate-I data, but is the N-tuple version which exhibits the smallest number of failures during the actual evaluation run (Estimate-II). The reason N-Version Programming has difficulty competing with the "best" version is that the selected N-tuples of low average reliability are composed of versions which are not "balanced", i.e. their reliabilities are very different and therefore variance of the average N-tuple reliability is large. As average N-tuple reliability increases N-Version Programming performance approaches, or exceeds, that of the "best" version. In part, this is because N-tuples become more "balanced" since the number of higher reliability versions in the subpool from which versions are selected is limited. This effect is further discussed in the text related to Table 3 and Figure 7. We also see that N > 3 improves performance of Consensus Voting more than it does that of Majority Voting. This is to a large extent because for N>3 plurality decisions become possible, i.e. in situations where there is a unique maximum of identical outputs the output corresponding to this maximum is selected as the correct answer even though it is not in majority.

Table 3 gives examples of the detailed behavior of selected individual N-tuples. In the table we first show the average reliability of the N-tuple (Avg. Rel.), its standard deviation (Std. Dev.), and the reliability of the acceptance test used in strategies that need it (AT Rel.). The table then shows the

average conditional voter decision space (CD-Space), and its standard deviation of the sample. Average conditional voter decision space is defined as the average size of the space (i.e. the number of available unique answers) in which the voter makes decisions given that at least one of the versions has failed. We use CD-Space to focus on the behavior of the voters when failures are present. Of course, the maximum voter decision space for a single test case is N. We then show the count of the number of times the "best" version in an N-tuple was correct (Best Version), and the success frequency under each of the investigated fault-tolerance strategies. The best response is underlined with a full line, while the second best with a wavy line.

Also shown in the table is the breakdown of the decision process for N-Version Programming with Consensus Voting (NVP-CV), i.e. the frequency of sub-events that yielded the consensus decision. We recorded the number of times consensus was a successful majority (S-Majority), an unsuccessful majority (F-Majority), a successful plurality (S-Plurality), an unsuccessful plurality (F-Plurality), a successful (S-Random) and an unsuccessful (F-Random) attempt at breaking a tie by random selection, and a failure by fiat (F-Fiat) by which we mean a situation where a tie existed but all the groups of outputs involved contained wrong answers so any choice made to break the tie led to failure. The sum of S-Majority, S-Plurality and S-Random comprises consensus voting success total, while the sum of F-Majority, F-Plurality, F-Random and F-Fiat is equal to the total number of cases where voting failed (F-Total).

Columns 1 and 2 of Table 3 show the results for two unbalanced low reliability 3-tuples, while column 3 shows the results for a well balanced 3-tuple of higher reliability. We see that in the former case the highest reliability is that of the best version while in the latter N-Version Programming with Consensus Voting offers the best result. An examination of Consensus Voting sub-events shows that in the case of 3-tuples most of the voting success came from majority agreements. The rest of the cases resulted in failures because all three versions returned different results. Consensus Voting attempts to salvage this situation. For instance, for the 3-tuple in column 1 Consensus Voting attempted to recover 293 times by random selection of one of the outputs. As would be expected, it succeeded about 30% of the time. Notice that in column 3 N-Version Programming with Consensus Voting is more successful than Consensus Recovery Block with Consensus Voting. This is because N-Version Programming with Consensus Voting five times successfully broke tie by random selection, while at the same time Consensus Recovery Block with Consensus Voting unsuccessfully acceptance tested the answers.

Columns 4-11 illustrate behavior of 5-tuples, and columns 12-15 behavior of 7-tuples. When $N > 3$ advantages of Consensus Voting over Majority Voting increase because plurality vote is now

possible. One problem that N-Version Programming with Majority Voting does not solve are the small space situations where the vote fails because a voter is offered more than two groups of answers from which to select the "correct" output, but there is no majority so voting cannot return a decision. The events are those where there is no agreement majority but one of the outputs occurs more frequently than any other, and those where there is a tie between the maximum number of outputs in two or more groups of outputs. For example, consider the 5-version system from column 4 where N-Version Programming with Consensus Voting is more successful than N-Version Programming with Majority Voting. Correct majority was available in only 321 cases, while in 146 instances the correct output was chosen by plurality. In comparison, the 3-version N-Version Programming with Consensus Voting system from column 1 is more successful than N-Version Programming with Majority Voting primarily because of the random selection process (S-Random).

Table 3. Examples of the frequency of voting and recovery events.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
							N-tuple Structure								
Versions	6,10,16	8,9,18	7,11,13	2,4,8, 12,16	6,9,12 13,14	4,7,8 11,13	1,3,5 8,20	5,7,11 13,20	4,5,7 8,11	3,5,10 11,17	4,5,8 12,20	2,3,5 7,11, 14,20	2,3,4 9,11, 13,20	1,3,8, 9,11, 15,16	5,6,8 10,12, 16,20
							Mean Value								
Avg. Rel.	0.59	0.75	0.91	0.67	0.58	0.86	0.70	0.86	0.86	0.92	0.71	0.84	0.83	0.61	0.63
Std. Dev.	0.36	0.21	0.03	0.26	0.21	0.12	0.20	0.11	0.13	0.05	0.21	0.13	0.13	0.22	0.25
AT Rel.	0.89	0.91	0.89	0.91	0.91	0.91	0.91	0.91	0.994	0.994	0.91	0.994	0.91	0.91	0.91
CD-Space	2.91	2.59	2.11	2.43	4.20	2.39	2.81	2.43	2.37	2.42	2.66	2.65	2.53	5.11	4.05
Std. Dev.	0.29	0.49	0.31	1.29	0.91	0.68	0.84	0.63	0.61	0.72	0.76	1.02	0.90	1.15	1.43
							Success Frequency								
Best Version	500	497	469	468	439	469	451	469	469	500	468	469	468	450	500
NVP-MV	206	341	486	321	285	468	405	468	454	466	430	465	455	280	282
NVP-CV	310	384	491	467	350	483	467	495	474	486	457	482	485	436	464
RB	443	461	443	454	454	454	441	463	478	493	458	485	462	441	458
CRB-MV	444	468	486	467	467	468	453	492	475	493	467	486	476	454	471
CRB-CV	444	468	486	467	465	481	466	494	475	493	467	482	484	433	467
							Success Frequency of Consensus Voting Sub-Events								
S-Majority	206	341	484	321	285	468	405	468	454	466	430	465	455	280	282
F-Majority	1	32	0	19	0	0	18	0	18	0	19	0	0	0	0
S-Plurality	0	0	0	146	42	13	61	18	14	16	23	17	29	146	171
F-Plurality	0	0	0	0	2	14	0	0	0	0	0	15	14	40	20
S-Random	104	43	5	0	23	2	1	9	6	4	4	0	1	10	11
F-Random	189	84	9	0	120	3	1	5	8	14	10	3	1	11	16
F-Flat	0	0	0	14	28	0	14	0	0	0	14	0	0	13	0
F-Total	190	116	9	33	150	17	33	5	26	14	43	18	15	64	36

Theoretical relations between voter decision space cardinality and voting strategies assuming failure independence is shown in Figure 5 for a simulated 5-version system composed of mutually independent versions with average N-tuple reliability of 0.85[6]. We plot system reliability of N-Version Programming with Consensus Voting and N-Version Programming with Majority Voting against the average conditional voter decision space. The average conditional voter decision space was calculated as the mean number of distinct results available to the voter during events where at least one of the 5-tuple versions has failed. The illustrated variation in the average conditional voter decision space (v) was obtained by varying the output space cardinality from r=2 to r=infinity. This resulted in the variation in v in the range $2 \leq v < 2.35$. Also shown is the N-Version Programming 2-out-of-N boundary (r = infinity).

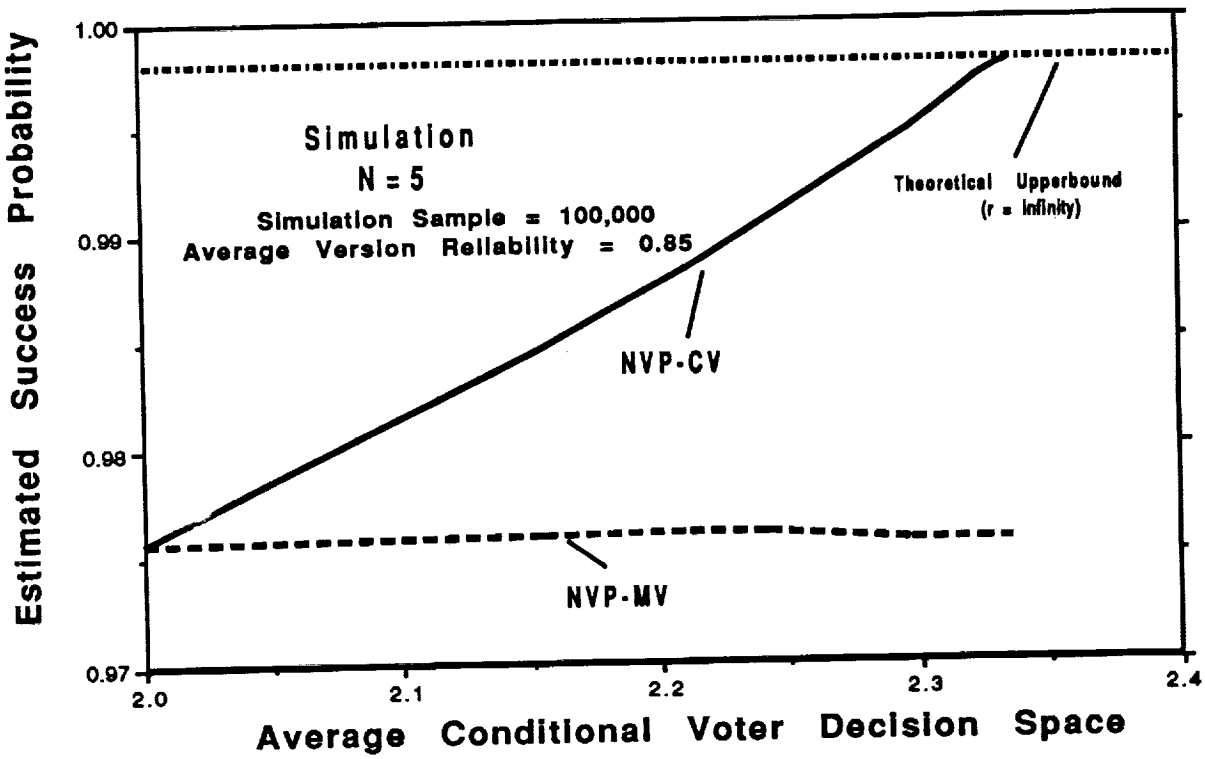


Figure 5. Influence of voter space size on different voting strategies.

Theory predicts that as the decision space increases ($v > 2$) the difference between the reliability of the systems using N-Version Programming with Consensus Voting and systems using N-Version Programming with Majority Voting increases in favor of N-Version Programming with Consensus

[6]Reliability of individual versions ranged between about 0.78 and 0.91, standard deviation of the sample was 0.061.

Voting [McA90]. Figure 6 illustrates the observed relationship between system success frequency and the average conditional voter decision space for a subset of 5-version systems with N-tuple reliability close to 0.85. Note that in Figure 6 the variation in the voter decision space size is caused by the variation in the probability of obtaining coincident but different incorrect answers. The observed behavior is in good general agreement with the trend shown in Figure 5 except that in the experiment, as the decision space increases, the reliability of N-Version Programming with Consensus Voting increases at a slower rate and reliability of N-Version Programming with Majority Voting appears to decrease.

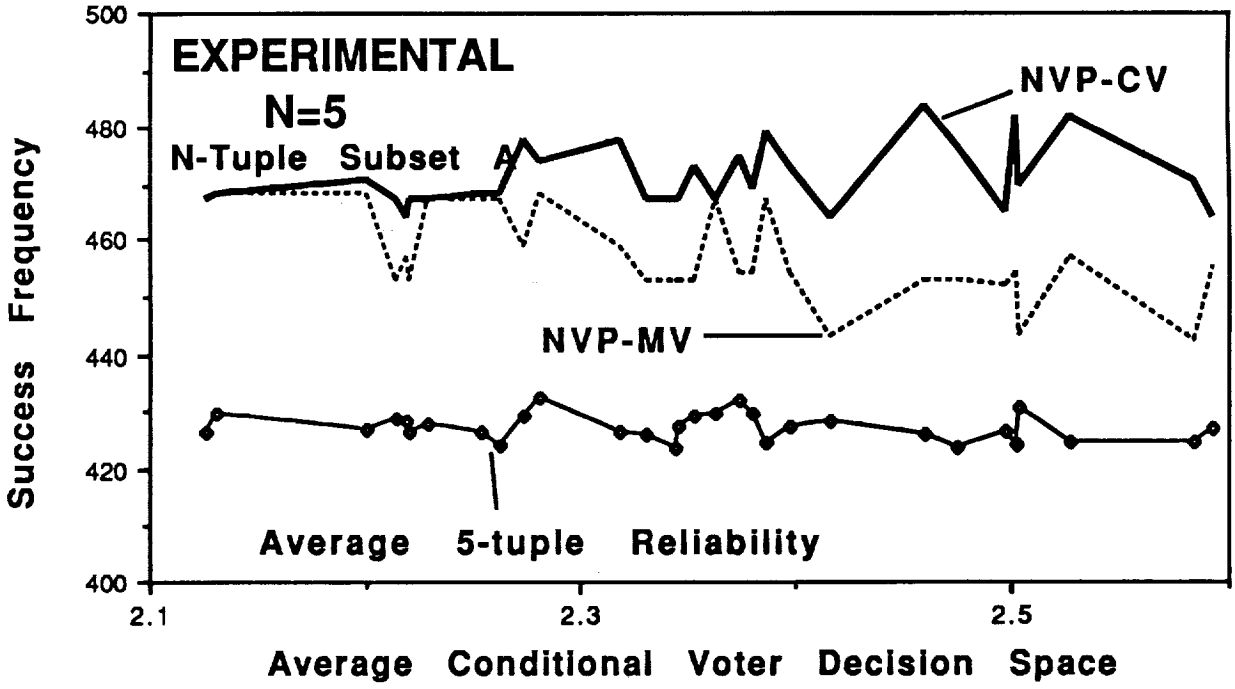


Figure 6. Voter behavior in small decision space.

In practice, failure probabilities of individual versions have a nonzero standard deviation about N-tuple mean. Small scatter may, up to a point, appear to increase average reliability obtained by voting because there may be enough versions on the "high" side of the mean to form a correct agreement number more often than would be expected from a set where all versions have the same reliability. But when the scatter is excessive the system reliability can actually be lower than the reliability of one or more of its best component versions [McA90].

This effect is illustrated in Figure 7 (independent model simulation; 100,000 cases for each point shown). In the figure we plot reliability of N-Version Programming with Consensus Voting and reliability of N-Version Programming with Majority Voting against the standard deviation of the N-tuple reliability (the mean value being constant and equal to 0.95). Also shown is the reliability of

the best single version obtained from the simulation. The feature to note is the very sharp step in the best version reliability once some critical value of the standard deviation of the sample is exceeded (about 0.03 in this example). The effect can be seen for some of the tuples shown in Table 3 (e.g. columns 1, 2, 4, 10, 11, and 15). Low average reliability systems with a high standard deviation about the mean tend to perform worse than the "best" version.

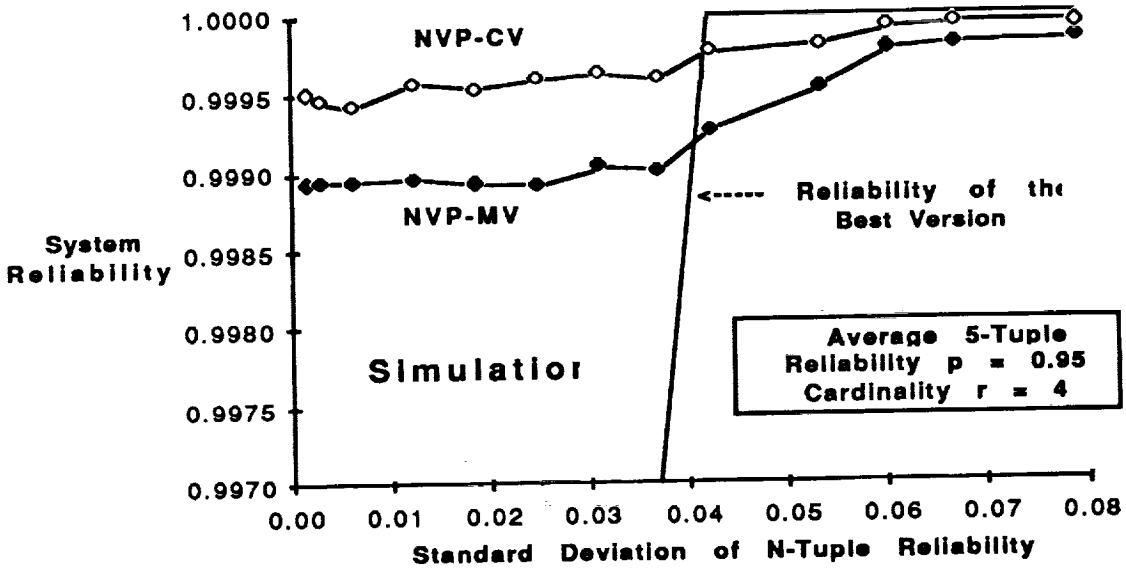


Figure 7. System reliability by Consensus Voting for 5-version systems vs. standard deviation of 5-tuple reliability. The probability of each j=2,..,r failure state is $\frac{1-\bar{p}}{r-1}$, where $\bar{p}$ is the average 5-tuple reliability.

A general conclusion regarding Consensus Voting is experimental results indicate that N-Version Programming with Consensus Voting appears to behave like its models based on failure independence predict. The advantage of Consensus Voting is that it is more stable than Majority Voting. It always offers reliability at least equivalent to Majority Voting, and it performs far better than Majority Voting when average N-tuple reliability is low or the average decision space in which voters work is large. When reliability is the issue N-Version Programming with Consensus Voting should be preferred to N-Version Programming with Majority Voting. A practical disadvantage of Consensus Voting may be the added complexity of the voting algorithm (or hardware) since the strategy requires multiple comparisons and random number generation.

3.3 Consensus Recovery Block

Theory predicts that in an ideal situation (version failure independence, zero probability for identical-and-wrong responses, perfect voter) Consensus Recovery Block is always superior to N-Version Programming (given the same version reliabilities and the same voting strategy) or

Recovery Block (given the same version and acceptance test reliabilities) [Sco87, Bel90]. This is illustrated in Figure 8 using 2-out-of-N voting. It is interesting to note the cross-over point between Recovery Block and N-Version Programming caused by the finite reliability of the Recovery Block acceptance test ($1-\beta = 0.9$). Of course, the behavior is modified when different voting strategies are used or if inter-version failure correlation is substantial.

Given the same voting strategy for Consensus Recovery Block and N-Version Programming, in the presence of very high inter-version failure correlation we would expect Consensus Recovery Block to do better than N-Version Programming only in situations where coincidentally failing versions return different results. We would not expect from Consensus Recovery Block more than to match N-Version Programming in situations where the probability of identical-and-wrong answers is very high since many decisions would be then made in a very small voting space and the Consensus Recovery Block acceptance test would be invoked only very infrequently.

The experimental results are shown in Figures 9, 10 and 11. The number of times that the result provided by a strategy was correct is plotted against the average N-tuple reliability. The same acceptance test version was used by Consensus Recovery Block and Recovery Block. From Figure 9 we see that for N=3 Consensus Recovery Block with Majority Voting provides reliability always equal to or larger than the reliability by N-Version Programming with Majority Voting (given the same versions). The behavior of the same 5-version systems using Consensus Voting instead of Majority Voting is shown in Figures 10 and 11. From Figure 10a we see that with Consensus Voting N-Version Programming becomes almost as good as Consensus Recovery Block at lower N-tuple reliabilities than is the case with Majority Voting. Figure 10b shows that Consensus Recovery Block with Consensus Voting is quite successful in competing with the "best" version. We also see that the expected cross-over point between N-Version Programming and Recovery Block is present, and that reliability of Consensus Recovery Block with Consensus, or Majority, Voting is usually at least as good as that by Recovery Block (Figures 9, 10b). However, it must be noted that given a sufficiently reliable acceptance test, or binary output space, or very high inter-version failure correlation, all the schemes that vote may have difficulty competing with Recovery Block. Also observed were two less obvious events described below. Both stem from the difference between the way Consensus Voting is implemented with N-Version Programming and the way it is implemented when used in Consensus Recovery Block.

Although Consensus Recovery Block with Consensus Voting is a more advanced strategy than N-Version Programming with Consensus Voting, and is usually more reliable than N-Version Programming with Consensus Voting, there are situations where the reverse is true. Because

Consensus Recovery Block with Consensus Voting employs the acceptance test to resolve situations where there is no plurality while N-Version Programming with Consensus Voting uses random tie breaking, occasionally N-Version Programming with Consensus Voting may be marginally more reliable than Consensus Recovery Block with Consensus Voting. This will happen when the acceptance test reliability is low, or when acceptance test and program failures are identical-and-wrong. Examples of this behavior can be seen in columns 3, 6, 7, 8 and 13 of Table 3. The difference in favor of N-Version Programming with Consensus Voting is often exactly equal to S-Random.

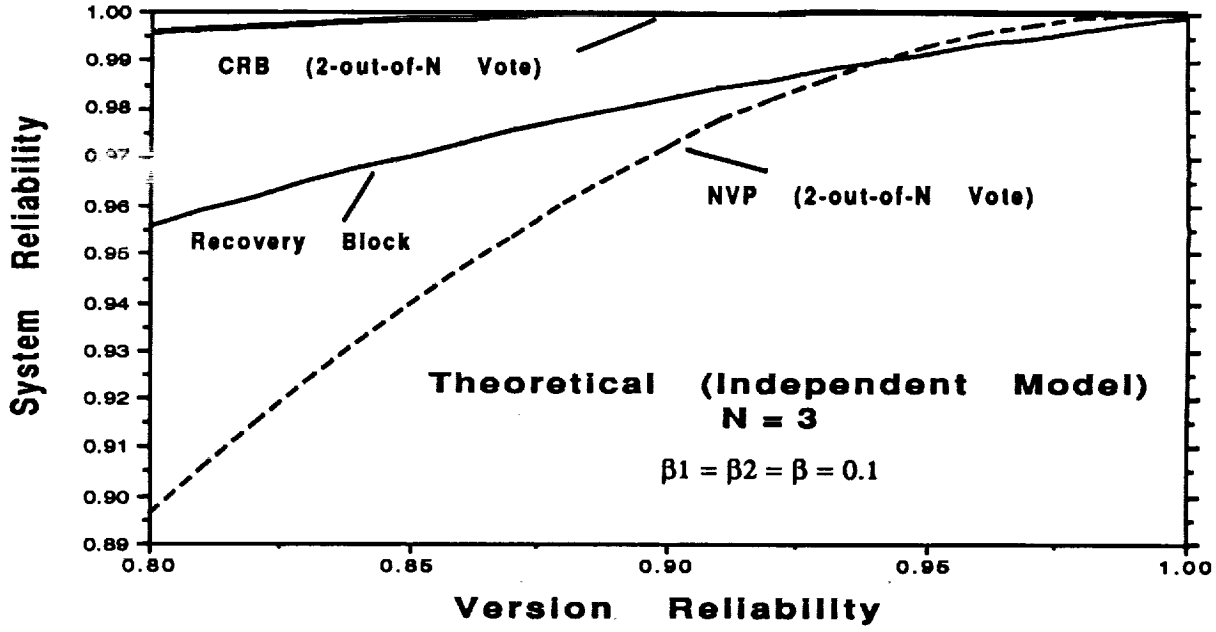


Figure 8. System reliability for different software fault tolerance schemes with 2-out-of-N voting, N = 3, and $\beta_1 = \beta_2 = \beta = 0.1$ (see Figure 1).

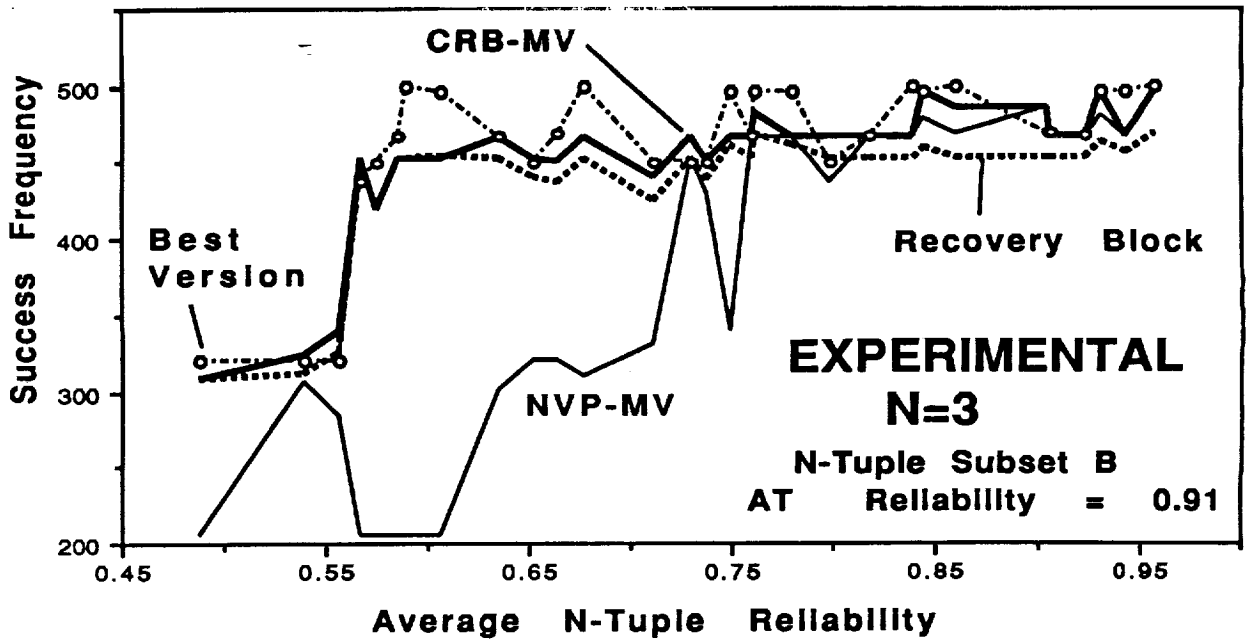


Figure 9. Consensus Recovery Block system reliability with majority voting.

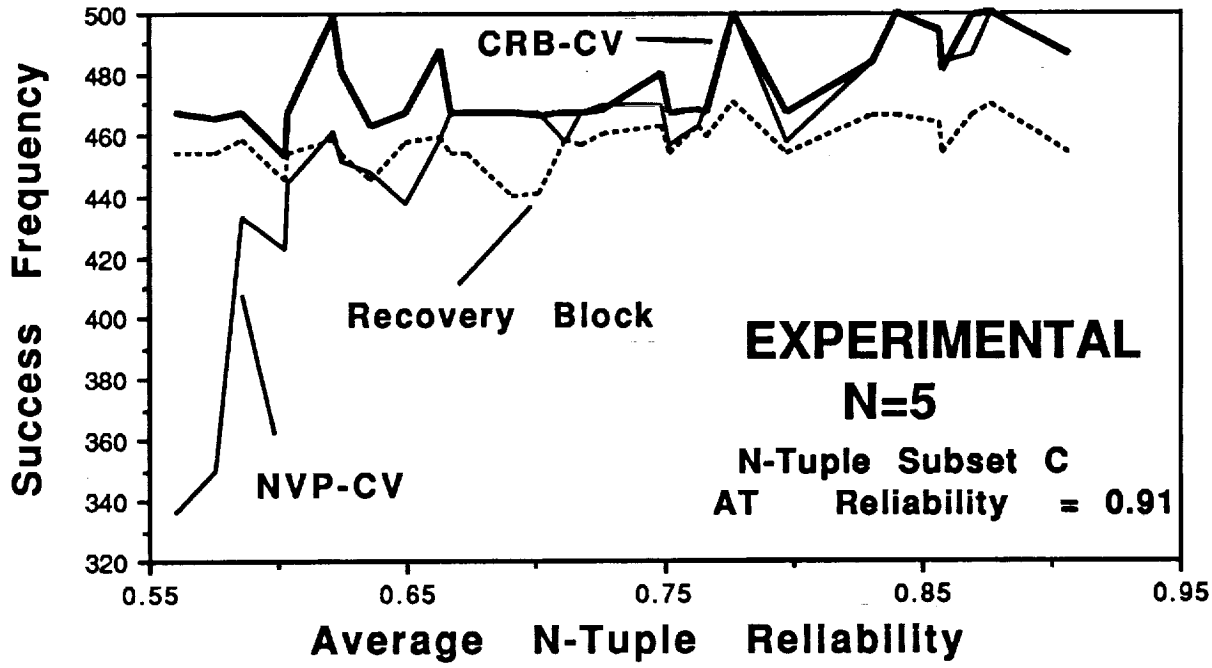


Figure 10a. Consensus Recovery Block with Consensus Voting compared with N-Version Programming with Consensus Voting and Recovery Block.

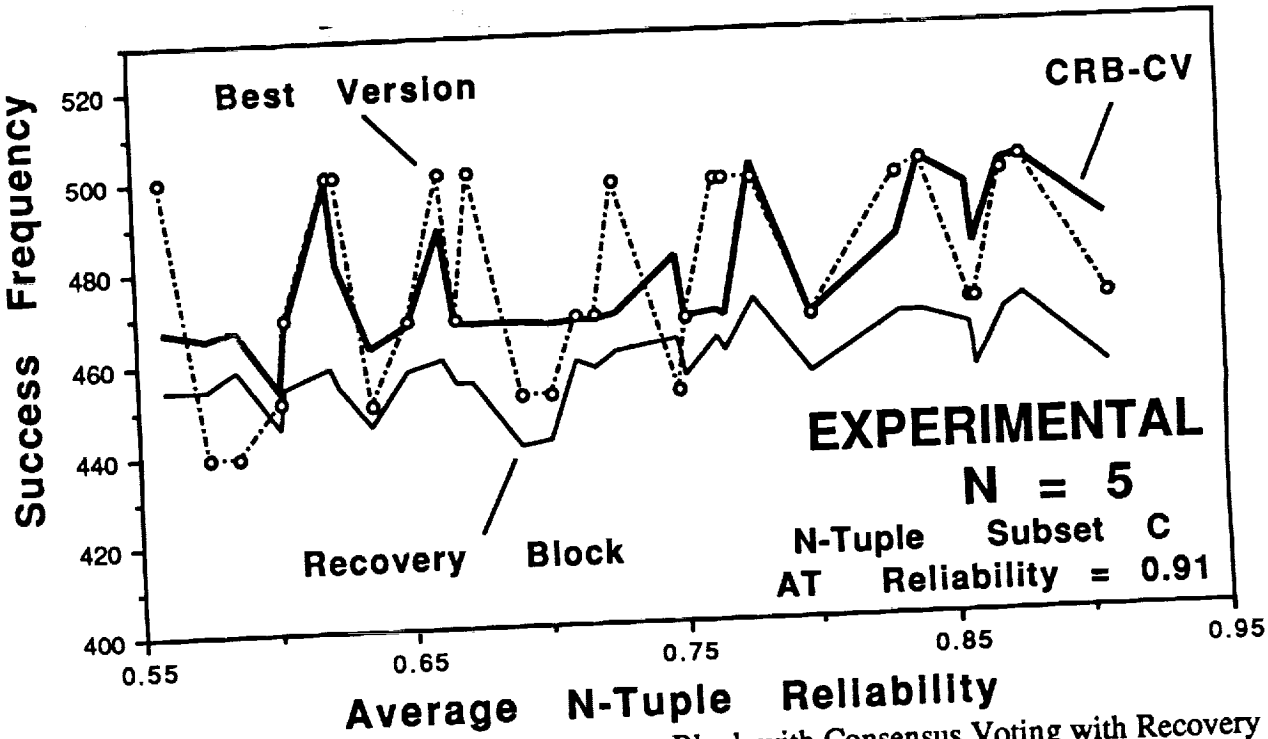


Figure 10b. Comparison of Consensus Recovery Block with Consensus Voting with Recovery Block and best version successes.

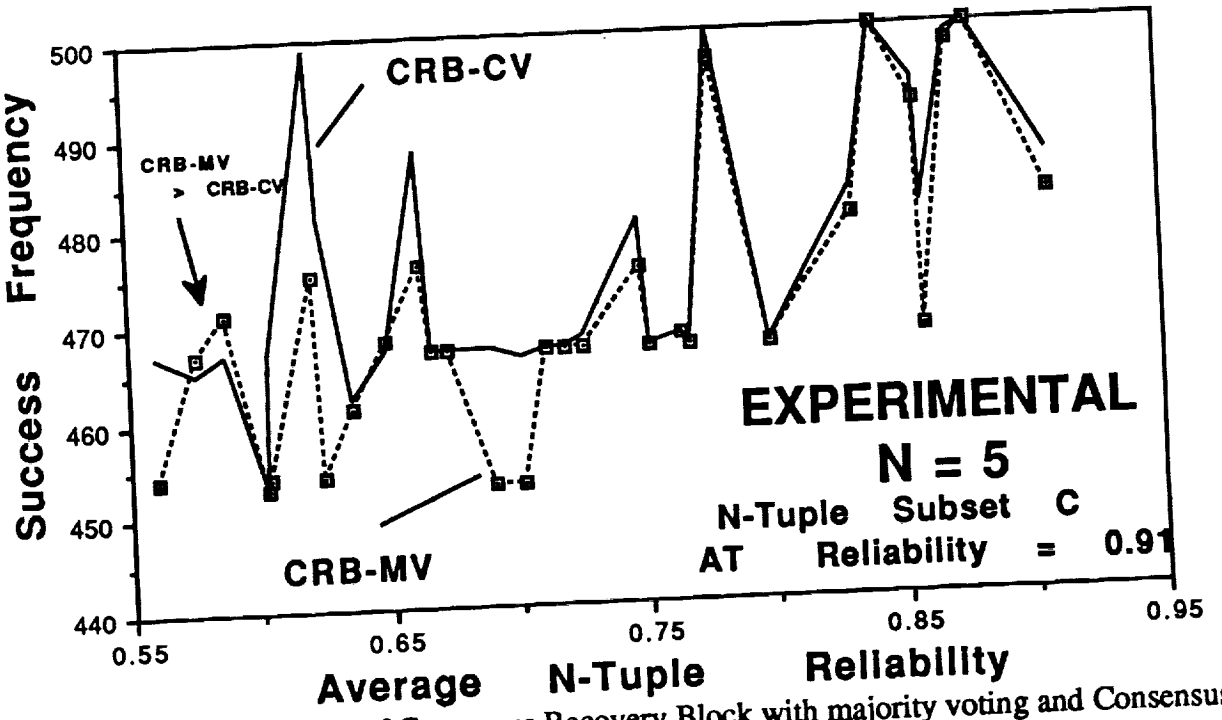


Figure 11. Comparison of Consensus Recovery Block with majority voting and Consensus Recovery Block with Consensus Voting strategies.

Similarly, Consensus Recovery Block with Consensus Voting is usually more reliable than Consensus Recovery Block with Majority Voting. However, if the number of agreeing versions is less than the majority sometimes the reverse may be true. For instance, if there is no majority then Majority Voting will fail and pass the decision to the acceptance test (which may succeed), while Consensus Voting will vote and, if the plurality is incorrect because of identical and wrong answers, Consensus Voting may return an incorrect answer. An example can be found in Figure 11 and in columns 5, 12, 14 and 15 of Table 3.

A general conclusion regarding the observed Consensus Recovery Block implementations is that the strategy appears to be quite robust in the presence of high inter-version correlation, and that the behavior is in good agreement with analytical considerations based on models that make the assumption of failure independence [Sco87, Bel90]. Of course, the exact behavior of a particular system is more difficult to predict since correlation effects are not part of the models. An advantage of Consensus Recovery Block with Majority Voting is that the algorithm is far more stable and is almost always more reliable than N-Version Programming with Majority Voting. But, the advantage of using a more sophisticated voting strategy such as Consensus Voting, may be marginal since the Consensus Recovery Block version of the Consensus Voting algorithm relies on the acceptance test to resolve ties. However, Consensus Voting version of CRB may be a better choice in high correlation situations where the acceptance test is of poor quality. In addition, Consensus Recovery Block will perform poorly in all situations where the voter is likely to select a set of identical-and-wrong responses as the correct answer (binary voting space). To counteract this we could either use a different mechanism such as the Acceptance Voting algorithm or an even more complex hybrid mechanism which would run Consensus Recovery Block and Acceptance Voting in parallel and adjudicate series-averaged responses from the two [Ath89, Bel90]. A general disadvantage of all hybrid strategies is an increased complexity of the fault-tolerance mechanism.

4. Summary and Conclusions

In this paper we presented the first experimental evaluation of Consensus Voting, and an experimental evaluation of the Consensus Recovery Block scheme. The evaluations were performed under conditions of high inter-version failure correlation and version reliability in the range between about 0.5 and 0.99.

The experimental results confirm the superior reliability performance of Consensus Voting over Majority Voting. They also confirm that Consensus Recovery Block strategy outperforms simple N-Version Programming and is very robust in the presence of inter-version failure correlation. In

general, the experimental results agree very well with the behavior expected on the basis of analytical studies of the hybrid models. Of course, behavior of an individual practical system can deviate considerably from that based on its theoretical model average, and so considerable caution is needed when predicting behavior of practical fault-tolerant software systems particularly if presence of inter-version failure correlation is suspected.

References

[Ath89] A.M. Athavale, "Performance Evaluation of Hybrid Voting Schemes," M.S. Thesis, North Carolina State University, Department of Computer Science, 1989.

[Avi77] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution", Proc. COMPSAC 77, 149-155, 1977.

[Avi85] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," IEEE Trans. Soft. Eng., Vol. SE-11 (12), 1491-1501, 1985.

[Bel90] F. Belli and P. Jedrzejowicz, "Fault-Tolerant Programs and Their Reliability," IEEE Trans. Rel., Vol. 29(2), 184-192, 1990.

[Bel91] F. Belli and P. Jedrzejowicz, "Comparative Analysis of Concurrent Fault-Tolerance Techniques for Real-Time Applications", Proc. Intl. Symposium on Software Reliability Engineering, Austin, TX, pp., 1991.

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes, P Humphreys, G. Dahl, and J. Lahti, "PODS--A Project on Diverse Software", IEEE Trans. Soft. Eng., Vol. SE-12(9), 929-940, 1986.

[Deb86] A.K. Deb, and A.L. Goel, "Model for Execution Time Behavior of a Recovery Block,", Proc. COMPSAC 86, 497-502, 1986.

[Deb88] A.K. Deb, "Stochastic Modelling for Execution Time and Reliability of Fault-Tolerant Programs Using Recovery Block and N-Version Schemes," Ph.D. Thesis, Syracuse University, 1988.

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Eck91] D.E. Eckhardt, A.K. Caglayan, J.P.J. Kelly, J.C. Knight, L.D. Lee, D.F. McAllister, and M.A. Vouk, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," IEEE Trans. Soft. Eng., Vol. 17(7), pp 692-702, 1991.

[Grn80] A. Grnarov, J. Arlat, and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies," Proc. FTCS 10, pp 251-253, 1980.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multi-version Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[Lap84] J.-C. Laprie, "Dependability Evaluation of Software Systems in Operation," IEEE Trans. Soft. Eng., Vol. SE-10 (6), 701-714, 1984.

[Lit87] B. Littlewood, and D.R. Miller, "A Conceptual Model of Multi-Version Software," FTCS 17, Digest of Papers, IEEE Comp. Soc. Press, pp 150-155, July 1987.

[Lit90] B. Littlewood and D.R. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software," IEEE Trans. Soft. Eng., Vol. 15(12), 1596-1614, 1989.

[McA90] D.F. McAllister, C.E. Sun and M.A. Vouk, "Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces", IEEE Trans. Rel., Vol 39(5), pp 524-534, 1990.

[Nic90] V.F. Nicola, and Ambuj Goyal, "Modeling of Correlated Failures and Community Error Recovery in Multi-version Software," IEEE Trans. Soft. Eng., Vol. 16(3), pp, 1990.

[Par90] A.M. Paradkar, "Performance Analysis of Multi-Stage N-Version Fault-Tolerant Software," M.S. Thesis, North Carolina State University, 1990.

[Ran75] B. Randell, "System structure for software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE-1, 220-232, 1975.

[Sco84a] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", IEEE FTCS 14,1984

[Sco84b] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984

[Sco87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., Vol SE-13, 582-592, 1987.

[Shi88] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault Elimination," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 180-187, July 1988.

[Tri82] K.S. Trivedi, "Probability and Statistics with Reliability, Queueing, and Computer Science Applications, Prentice-Hall, New Jersey, 1982.

[Tso86] K.S. Tso, A. Avizienis, and J.P.J. Kelly, "Error Recovery in Multi-Version Software," Proc. IFAC SAFECOMP '86, Sarlat, France, 35-41, 1986.

[Tso87] K.S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation", Proc. IEEE 17th Fault-Tolerant Computing Symposium, pp 127-133, 1987.

[Vou90] Vouk, M.A., Caglayan, A., Eckhardt D.E., Kelly, J., Knight, J., McAllister, D., Walker, L., "Analysis of faults detected in a large-scale multiversion software development experiment," Proc. DASC '90, pp 378-385, 1990.

Paper Appendix I

To select subsets of N-tuples with have certain properties such as approximately equal reliabilities we used the following approach.

We first select acceptance test versions based on Estimate I data (for example, one low reliability, one medium and one high reliability acceptance test). These versions are then removed from the pool of 20 versions. Also removed from the pool might be versions which have either very low or very high reliability to better balance reliabilities of the selected N-tuples. For a given N the remainder of the versions (a subpool) are then randomly sampled without replacement until an N-tuple which has not already been accepted for the subset is formed. The average N-tuple reliability is then computed, and if it lies within the desired reliability range the N-tuple becomes a member the subset. Then the N-tuple versions are returned to the subpool and the next N-tuple is selected in a similar manner, etc. Once the subset contains either all possible combinations, or at least 600 N-tuples (whichever comes first), the subset is sorted by the average N-tuple reliability and standard deviation of the average N-tuple reliability.

If a single reliability category is desired (e.g. between 0.8 and 0.9) then the first 30 versions with the smallest N-tuple standard deviation are chosen and run in the experiment.

If a range of reliabilities is desired, the range is divided into categories in such a way that members of the same category have identical first two digits after the decimal point. Then from each category we chose the combination that has the smallest standard deviation of the average N-tuple reliability.

We have thus selected a number of subsets. The following are mentioned in the text

N-Tuple Subset A (5 version systems, average 5-tuple reliability in the range 0.8 to 0.9, acceptance test reliabilities of 0.67 (version 20), 0.93 (version 2) and 0.994 (version 18). Version 10 was not used (too reliable).

N-Tuple Subset B (3 version systems, average 3-tuple reliability in the range 0.5 to 1.0, acceptance test reliabilities of 0.67 (version 20), 0.89 (version 3) and 0.91 (version 17). Version 10 was used.

N-Tuple Subsets C and D were chosen in a manner similar to set B except that they consisted of 5-tuples and 7-tuples, respectively.

N-Tuple Subset E (5 version systems, average 5-tuple reliability about 0.91, no acceptance test)

Appendix VI:
Cost Modelling of Fault-Tolerant Software

D.F. Mcallister and R.K. Scott

(Published in Information and Software Technology, Vol 33 (8), pp 594-603, October 1991)

Cost modelling of fault-tolerant software

D F McAllister and R K Scott*

Costs of a simplex or single-version system are compared with the following three-version fault-tolerant software systems: N-version programming (NVP), recovery block (RB), and consensus recovery block (CRB). Cost is minimized subject to a system reliability constraint. The objective function of the optimization program is of the form $\sum \beta_i/(1-r_i)^{\alpha_i}$, *where the constants* β_i *and* α_i *are fixed and the* r_i *are variables that are reliabilities of the versions, the acceptance test in the case of RB and CRB and the voter in the case of NVP and CRB. The costs are compared for different values of* β_i *and values of* $\alpha_i = 0.5$, *1, and 2. Assuming that failures are independent, CRB followed by RB are the most cost-justifiable fault-tolerant techniques considered. Unless the voter is perfect, NVP does not compete cost-wise with the other two methods. Indeed, in some cases it is worse than a simplex system.*

cost modelling, fault-tolerant software, system reliability

There have been some attempts to model the cost of multiversion fault-tolerant software. Saglietti and Ehrenberger[1] treated the problem of estimating Poisson arrival rates of failure inputs to determine when it might be more cost-effective to devote testing time to a single-version versus a two-version system. Laprie *et al.*[2] presented a simple cost model for the N-version programming and recovery block fault-tolerant software systems. Their model is used to estimate values of parameters in the models presented here. Scott *et al.*[3-6] introduced data domain reliability models of several fault-tolerant software schemes, including N-version programming (NVP), recovery block (RB), and consensus recovery block (CRB). These models are extended by coupling them with a cost function and the results examined when the cost is constrained by system reliability.

Scott *et al.*[3] were first to show that failures can be correlated in independently developed, functionally equivalent software versions, and they developed models that could be used in treating this correlation in predicting system reliability. The existence of correlated failures was corroborated in an experiment by Knight *et al.*[7] and again by Kelly *et al.*[8]. To treat the failure correlation, Arlat *et al.*[9] presented Markov models based on the identification of fault types and an analysis of the behaviour of fault-activation. The model allows for positive correlation among faults. Eckhardt and Lee[10] presented another model for the analysis of coincident failures in multiversion software.

If failures are dependent, Scott *et al.*'s models[3] require estimation of conditional probabilities, which significantly increases the parameters in a reliability model. Hence, for tractability the authors restrict their development to three-version systems and assume that software failures are statistically independent, i.e., that failures are not correlated. The authors' results will provide a lower bound on costs when failures are correlated as version, voter, and acceptance test reliabilities must be higher to meet a system reliability constraint that will increase system costs.

The reliability of a software module is the probability that it produces the correct result for a given input. The authors' notation will be consistent with their previous work. Let r_1, r_2, and r_3 be reliabilities of each version of a three-version fault-tolerant system, let B be the reliability of the acceptance test in RB and CRB, let V be the reliability of the voter in NVP and CRB, and let S be the system reliability. Then for NVP:

$$S_{NVP}(r_1,r_2,r_3,V) = V(r_1r_2 + r_1r_3 + r_2r_3 - 2r_1r_2r_3) \qquad (1)$$

In RB, it is assumed that the probability of rejecting a correct answer is equal to the probability of accepting an incorrect one. In this case RB becomes:

$$\begin{aligned} S_{RB}(r_1,r_2,r_3,B) = B(&r_1 + r_1r_2 + r_1r_3 + r_2B - 2r_1r_2B + r_1r_3B \\ &+ r_2r_3B - 4r_1r_2r_3B + r_3B^2 - 2r_1r_3B^2 \\ &- 2r_2r_3B^2 + 4r_1r_2r_3B^2) \end{aligned} \qquad (2)$$

while CRB is defined by:

$$\begin{aligned} S_{CRB}(r_1,r_2,r_3,B,V) = {} & S_{RB}(r_1,r_2,r_3,B) + S_{NVP}(r_1,r_2,r_3,V) - \\ & S_{RB}(r_1,r_2,r_3,B)\, S_{NVP}(r_1,r_2,r_3,V) \end{aligned} \qquad (3)$$

While the equations tend to become visually complicated, they are simple to treat using a symbol manipulation program such as Mathematica[11]. In addition, some simplifying assumptions will be made for tractability and understanding.

The optimization problem of minimizing systems cost will be treated subject to the constraint that system reliability is fixed. The nonlinear optimization problem becomes:

$$\begin{aligned} &\text{Minimize } C(r_1,r_2,r_3,B,V) \qquad (O) \\ &\text{subject to the constraint} \\ &S(r_1,r_2,r_3,B,V) = R. \end{aligned}$$

Since reliabilities are probabilities, there are the addi-

Department of Computer Science, North Carolina State University, Raleigh, NC 27695 8206, USA.
*IBM, PO Box 12195, Research Triangle Park, NC 27709, USA

tional constraints that the r_i, B, V, and R must lie between 0 and 1.

The next section discusses the choice of a cost function. Then a special, easily solved subcase of the constrained optimization problem is treated, followed by treatment of a more general version of the optimization problem, solved using Lagrange multipliers. Finally, the results are summarized.

COST FUNCTION

It is assumed that the development cost increases exponentially as the reliability of a version approaches 1. This follows directly from data domain reliability modelling[5] as adding a correct digit to the reliability estimate of a software module requires an order-of-magnitude more test cases if random testing is used. In addition, the cost function should have the line $r = 1$ as a vertical asymptote. There are many choices for a cost function with the above properties, and the techniques proposed here can be applied to others also. The authors have chosen the cost function for a single version to be:

$$C(r) = \beta(1-r)^{-\gamma} + c$$

where r is the reliability of a version, and α, β, and c are positive constants that control the shape and location of the cost function. The constants β and c determine the initial or start-up cost when $r = 0$. Since the optimization results are independent of the constant c, which appears linearly in the equation, it will be eliminated from the definition of C henceforth. The final cost can be augmented by c without changing the optimal reliabilities.

The constant α controls the rate at which the cost increases as r approaches 1, and the constant β can be used to control the initial cost and differences between development and testing costs of each module. In the most general case, each version, the acceptance test, and the voter can have different values of α, β with different reliabilities. To reduce the dimensionality of the problem, attention is restricted here to the case when all versions have the same reliability r and α is the same for all components, including the voter and the acceptance test.

In the next section it is assumed that the reliabilities of the acceptance test (B) and the voter (V) are equal to r. The cost function in this case becomes $\sum\beta_i/(1-r)^{\alpha}$ and hence the sum of the βs is just a multiplicative constant. The worst case is assumed where $\sum\beta_i$ is the number of modules involved in the fault-tolerant system (including the acceptance test and a voter).

In the section after that the constraint will be relaxed that the reliabilities of the voter and the acceptance test must be the same as the versions. There are parts of the development cycle that are common to all versions, such as the writing of the specification, and testing can be done in parallel using such techniques as back-to-back testing as recommended by Saglietti and Ehrenberger[11] and Vouk[12]. It is to be expected that the β values of the acceptance test (β_B) and the voter (β_V) will be less than or equal to the β values of the versions because, in general, an acceptance test and a voter should be less complex to write and more easily tested than any of the versions. Two subcases will be considered:

- the β_i of the versions are equal to 1
- the β_i of the versions are decreasing in accordance with the cost relationships proposed by Laprie *et al.*[2].

The authors will examine the behaviour of the cost function for different values of α and try to summarize the results and impart some intuition. The next section first treats the special case when:

$$r_1 = r_2 = r_3 = B = V$$

This reduces the above optimization problem to a straightforward root-finding problem for functions of a single variable. It is more tractable than the general case and provides useful bounds.

Table 1. Cost of three-version NVP assuming perfect voter and no cost ($\beta_V = 1$, $\beta_V = 0$)

		$\alpha = 0.5$		$\alpha = 1$		$\alpha = 2$	
R	$r(R)$	$C(R)$	$U(R)$	$C(R)$	$U(R)$	$C(R)$	$U(R)$
0.9	0.804200	6.8	3.2	15.3	10	78.2	100
0.99	0.941097	12.3	10	50.9	100	864.7	10000
0.999	0.981630	22.1	31.6	163.3	1000	8890	1000000
0.9999	0.994215	39.4	100	518.5	10000	89642	1 E 08
0.99999	0.998173	70.2	316.2	1642.0	100000	898760	1 E 10

MINIMIZING COST SUBJECT TO RELIABILITY CONSTRAINT

First the case is treated where all exponents α are equal and all reliabilities are constrained to be equal.

N-version programming

Since the model of NVP of Scott *et al.*[3] does not include an acceptance test and assumes a perfect voter with no cost ($V = 1$ and $\beta_V = 0$), it will be treated first. As it is being assumed that $r_1 = r_2 = r_3 = r$ and $\beta_1 = \beta_2 = \beta_3 = \beta_r = 1$, the cost function becomes ($C(r) = 3(1-r)^{-\alpha}$ and the system reliability is:

$$S_{NVP}(r) = 3r^2 - 2r^3 \tag{4}$$

The function $S_{NVP}(r)$ is monotone on the interval [0,1] and hence the equation $S_{NVP}(r) - R = 0$ has a single real root, denoted by $r(R)$, in [0,1]. In this case the optimal cost is:

$$C(r(R)) = 3/(1-r(R))^{\alpha} \tag{5}$$

As it is assumed that $\beta_r = 1$, the cost of a system with a single or unit version with the same system reliability is:

$$U(R) = 1/(1-R)^{\alpha} \tag{5}$$

The right-hand side of equation (5) is monotone increasing in R. Table 1 presents its values for $\alpha = 0.5$, 1, and 2

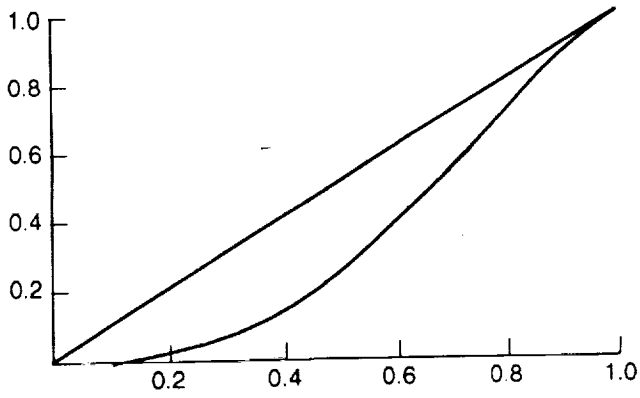


Figure 1. Graph of $S_{NVP}(r)$ with imperfect voter and V = r

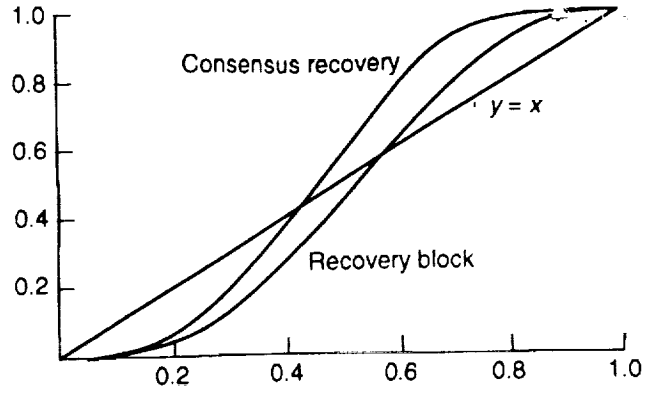


Figure 2. Reliability of RB and CRB assuming r = B = V

Table 2. Cost of three-version NVP assuming nonperfect voter ($\beta_r = \beta_v = 1$)

R	$r(R)$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 2$
0.9	0.917647	13.9	48.6	589.8
0.99	0.990279	40.6	411.5	42329
0.999	0.999003	126.7	4012	4.0 E 6
0.9999	0.999900	400	40000	4 E 8
0.99999	0.999990	1264.9	4 E 5	4 E 10

for R = 0.9, 0.99, 0.999, 0.9999, and 0.99999. As would be expected, the value of α is critical in drawing conclusions when comparing the cost of a single-version versus a three-version fault-tolerant system. When high system reliability is required it is more likely that a three-version system will be more cost effective than a single-version system in the case that the voter is perfect and has zero cost.

It is now assumed that the voter is neither perfect nor cost free. Let V be the reliability of the voter and $\beta_v = 1$. The model for the reliability of a three-version system where all the version reliabilities are equal becomes:

$$S_{NVP}(V,r) = V(3r^2 - 2r^3) \tag{6}$$

Assuming $\alpha_v = \alpha_r$ and $V = r$, the reliability constraint becomes $R = 3r^3 - 2r^4$. If the function $3r^3 - 2r^4$ for $0 \leqslant r \leqslant 1$ is graphed, it is found to be monotone and lies below the line $y = r$ (see Figure 1). The cost for this model is $C(r(R)) = (3 + \beta_v)/(1 - r(R))^\alpha$. Table 2 assumes that $\beta_v = 1$, hence $C(r(R)) = 4/(1 - r(R))^\alpha$. The cost of a unit version is the same as that given in Table 1 and is omitted.

Note that the imperfect voter causes $r(R)$ to be larger for each R compared to the perfect voter case. A larger $r(R)$ implies a greater cost per version. It is clear that a simplex system with the same system reliability will now be less costly because $r(R) \approx R$ for R close to 1. As the same is being paid (in terms of α) for the voter as for the versions, and it is assumed that the development of n versions is n times the cost of the development of a single version, it can be the case that an imperfect voter 3MR software system will be less cost effective than a simplex system. While this appears to be a startling result, it is due to the assumption of equal development costs (equal βs and αs). It will be seen that when the constraint that $V = r$ is removed and the βs are allowed to decrease NVP becomes more competitive.

Table 3. Cost of RB ($B = r$, $\alpha_B = \alpha_r$)

R	$r(R)$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 2$
0.9	0.790108	8.7	19.0	90.8
0.99	0.922202	14.3	51.4	660.9
0.999	0.971875	23.8	142	5047.8
0.9999	0.990438	40.9	418.3	43748
0.99999	0.996886	71.7	1284.5	4.1 E 5

Recovery block

If $B = r$ then the system reliability becomes:

$$S_{RB}(r) = 4r^6 - 8r^5 + 2r^4 + 2r^3 + r^2 \tag{7}$$

This sixth-degree polynomial is also monotone in [0,1], which implies $S_{RB}(r) - R$ has a single real root in [0,1]. Figure 2 includes a graph of $S_{RB}(r)$. Again assuming that $\beta_B = \beta_r = 1$ and $\alpha_B = \alpha_r$, this gives Table 3, which provides the values of $C(R) = 4/(1 - r(R))^\alpha$. Comparing Tables 1 and 3, it is clear that RB is more cost effective than three-version NVP with a perfect voter and is more cost effective than a single-version system for high-reliability cases.

Consensus recovery block

The reliability of the CRB[4] is given by equation (3). If $V = 1$, i.e., the voter has reliability 1, and $\beta_v = 0$, then $S_{NVP}(V,r)$ is given by equation (1). If it is also assumed that $B = r$ then $S_{RB}(r,B)$ is given by equation (3), this gives:

$$S_{CRB}(r) = 8r^9 - 28r^8 + 28r^7 + 2r^6 - 12r^5 - r^4 + 4r^2 \tag{8}$$

This function is monotone over [0,1]. Table 4 assumes that $\beta_B = 1$ and the voter has no cost. Hence $C(r(R)) = 4/(1 - r(R))^\alpha$. It is seen that CRB with a perfect voter is more cost effective than any of the previous systems, as

Table 4. Cost of CRB with perfect voter ($B = r$, $\beta_B = \beta_r = 1$, $\beta_V = 0$, and $\alpha_B = \alpha_r$)

R	$r(R)$	$\alpha=0.5$	$\alpha=1$	$\alpha=2$
0.9	0.632687	6.6	10.9	29.6
0.99	0.796570	8.9	19.7	96.7
0.999	0.882487	11.7	34	289.7
0.9999	0.931504	15.3	58.4	852.6
0.99999	0.960196	20	100.5	2524.7

Table 5. Cost of CRB ($V = B = r$, $\alpha_V = \alpha_B = \alpha_r$)

R	$r(R)$	$\alpha=0.5$	$\alpha=1$	$\alpha=2$
0.9	0.698340	9.1	16.6	54.9
0.99	0.847012	12.7	36.7	213.6
0.999	0.920723	17.8	63	795.6
0.9999	0.959660	24.9	123.9	3072.5
0.99999	0.980051	35.4	250.6	12563

would be expected, and the cost grows relatively slowly as R increases. For high-reliability systems, it will be significantly cheaper than a single-version system.

If equation (2) is used for NVP and equation (3) for RB then this gives:

$$S_{CRB}(r) = 8r^{10} - 28r^9 + 28r^8 - 2r^7 - 11r^6 + 5r^5 + r^2 \quad (9)$$

This is also monotonic on [0,1] (see Figure 2). Table 5 gives the values when all αs, rs, and V and B are equal. As it is assumed that $\beta_V = \beta_B = 1$ then the cost function is $C(R) = 5/(1-r(R))^2$.

Graphs of equations (7) and (9) are given in Figure 2.

From Tables 1 to 5 and the assumptions of this section, it is clear that the most cost-effective system in terms of total cost is CRB with a perfect voter followed by CRB with an imperfect voter. In general, CRB will be significantly less costly than either N-version programming, recovery block, or a single-version system. The least cost-effective system is NVP with an imperfect voter. Under the authors' assumptions it is even worse than a simplex system with the same reliability. As it has been assumed that all reliabilities are equal and all αs are equal these results provide an upper bound for the case that r, V, and B are not required to have equal reliabilities. This is demonstrated in the next section.

ELIMINATING EQUAL RELIABILITY CONSTRAINT $r = V = B$

This section treats the case when r, B, and V can have different values in the optimization problem (O). The cost will be minimized subject to the constraint that the system reliability must be met.

The authors have chosen to use the constrained optimization technique of Lagrange multipliers[15]. Applying this technique yields the following optimization problem. Let λ be a 'Lagrange multiplier', $C(x_1,x_2,...x_n)$ the objective function to be optimized, and let $G(x_1,x_2,...x_n) = K$ be the constraint. Form the function:

$$u = C(x_1,x_2,...,x_n) + \lambda G(x_1,x_2,...,x_n)$$

A solution $(x_1, x_2,...,x_n, \lambda)$ to the following set of nonlinear equations is an optimal solution to the original optimization problem:

$$\begin{aligned} \partial u/\partial x_1 &= 0, \\ \partial u/\partial x_2 &= 0, \\ &\vdots \\ \partial u/\partial x_n &= 0 \\ G(x_1, x_2,...,x_n) &= K \end{aligned}$$

The authors have applied unconstrained Newton's method for several variables[13] successfully for most cases of this problem. The requisite partial derivatives were calculated symbolically using Mathematica[11].

Some discussion of the numerical properties of the iterative technique is in order. Newton's method does not guarantee convergence for arbitrary starting values. Furthermore, convergence can occur at a point for which r, V, or B lies outside the allowable range, i.e., these values must be probabilities and lie in the interval [0,1]. Hence starting values are critical. The authors used a Pascal program called MINCOST, which runs on an IBM PC. All calculations were in double precision, which is approximately 14 decimal digits. The program allows the user to choose initial values for r, B, V, and λ. Newton's method uses a linearization of the nonlinear equations and solves the linearized version to calculate correction values to the current estimate of the solution. Once the correction values are sufficiently small or the number k of allowable iterations is exceeded the iteration is halted. If convergence has taken place and the values of r, B, or V lie outside the allowable range or if the number of iterations k has been exceeded then a search for a better starting value begins. This is accomplished by adding and subtracting a change value δ to each of r,V, and B until convergence in range takes place. If no convergence occurs for a given δ, then 2δ is tried and the search for convergence in range begins again. The process continues until convergence is achieved or a reliability lies outside [0,1]. If the system is used to find optimal values for several different Rs then arranging these values in ascending order, $R_1 < R_2 < ... < R_m$, and using the solutions for R_i as starting values for the optimization problem for R_{i+1} usually gives good results, especially if the R_i are 'close.'

For high system reliabilities of 0.9999 and 0.99999, numerical instabilities sometimes occurred. The instabilities manifested themselves when comparing costs for a given value of α for β values that were close (within 0.1). To correct these instabilities, the authors employed a technique called 'damped Newton'[16]. Instead of adding δ, the Newton correction vector, a multiple of δ is used. The multiple $(1/2)^j$, $j = 1, 2, 3,...$, is chosen so that the residual error decreases for successive iterations. When damped Newton was applied the instabilities disappeared. Note that j should not be chosen so large that the

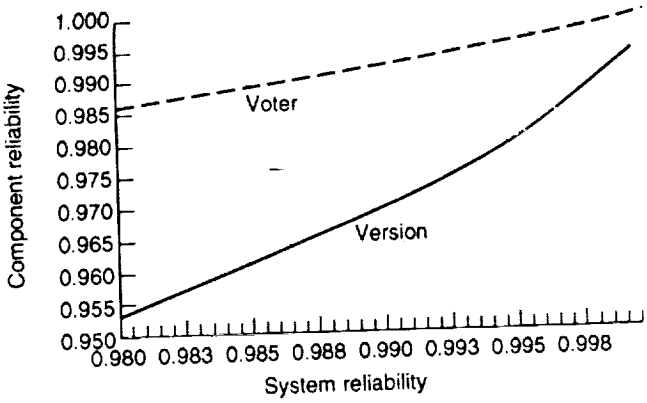


Figure 3. Plot for NVP of version, voter, and acceptance reliabilities for R = *0.98 to 0.99999,* β = *1,* α = *1*

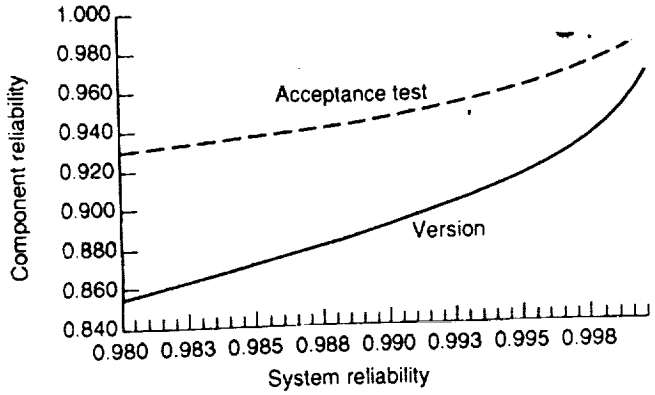


Figure 4. Plot for RB of version, voter, and acceptance reliabilities for R = *0.98 to 0.99999,* β = *1,* α = *1*

components of δ become smaller than the desired convergence criterion, or else the process will appear to converge prematurely.

N-version programming

Recall that if $r_1 = r_2 = r_3$, equation (6) gives the reliability:

$$S_{NVP}(r,V) = V(2r^3 - 3r^2)$$

Recovery block

If B is not equal to r then $S_{RB}(r,B)$ becomes the bicubic polynomial:

$$S_{RB}(r,B) = 4B^3r^3 - 4B^3r^2 + B^3r - 4B^2r^3 + B^2r + Br^3 + Br^2 + Br \quad (10)$$

Note that the surface $S_{RB}(r,B)$ is symmetric in r and B. As it is assumed that $r_1 = r_2 = r_3 = r$, the cost function is of the form $k/(1-r)^2 + \beta_B/(1-B)^2$. Hence it would be expected that if $k > \beta_B$, i.e., r is weighted more than B, which is usually the case, then $r < B$ in the optimal solution. It will be found that this is the case in the numerical results.

Consensus recovery block

$S_{CRB}(r,B,V)$ is given by equations (6) and (10) in terms of S_{NVP} and S_{RB} and equals:

$$r(B + B^2 + B^3 + Br - 4B^3r + Br^2 - 4B^2r^2 + 4B^3r^2 + 3rV - 2r^2V - 3Br^2V - 3B^2r^2V - 3B^3r^2V - Br^3V + 2B^2r^3V + 14B^3r^3V - Br^4V + 12B^2r^4V - 20B^3r^4V + 2Br^5V - 8B^2r^5V + 8B^3r^5V) \quad (11)$$

Numerical results

The single-variable case discussed previously is a relatively tight upper bound for the case when all β values are equal. Hence for quick approximations, assuming that all reliabilities, βs and αs are equal gives good results and is considerably more tractable. The minimum costs were computed for several combinations of βs and αs for the system reliabilities 0.9, 0.99, 0.999, 0.9999, and 0.99999.

Figures 3–5 are plots for NVP, RB, and CRB of the

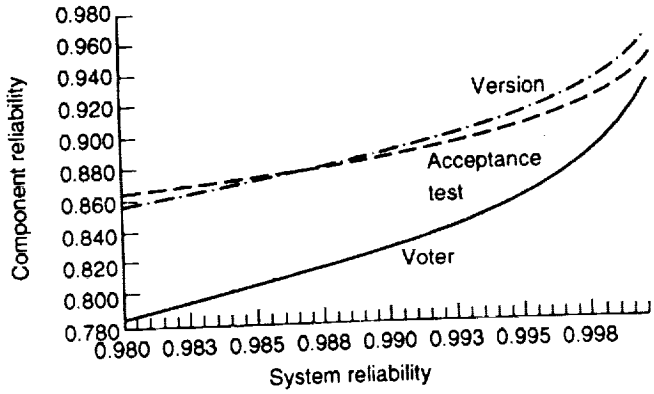


Figure 5. Plot for CRB of version, voter, and acceptance reliabilities for R = *0.98 to 0.99999,* β = 1, α = 1

version, voter, and acceptance reliabilities from $R = 0.98$ to $R = 0.99999$ in the optimal solution for the case that the βs are 1 and the αs are equal and set to 1. The cases for α = 0.5 and α = 2 are similar. Note that in Figure 3 the voter must be considerably more reliable than the versions and that a simplex system is less costly by a factor of 2 to 3. As shown in Figure 4, the acceptance test must also be more reliable than the versions, but by at most 0.05. The RB is less costly than a simplex system in this range, with the difference in costs increasing as higher system reliability is required. The CRB is also less costly than a simplex system. Figure 5 shows that the acceptance test must be more reliable than either the voter or the versions, and there is a crossover point where the voter must be more reliable than the versions. The cost ratio between CRB and RB is approximately 0.5 to 0.6.

For α = 0.5, a simplex system was less costly than either RB or CRB until system reliability was above 0.99. This was not true for α = 1 or α = 2. This implies that as α increases, both RB and CRB will be relatively less costly for high system reliability. Figure 6 shows the optimal cost versus system reliability of each of the fault-tolerant techniques considered.

As the objective function is linear in the constants β_i, for a given fault-tolerant technique holding the αs fixed will result in identical optimal values for r, V, and B for a

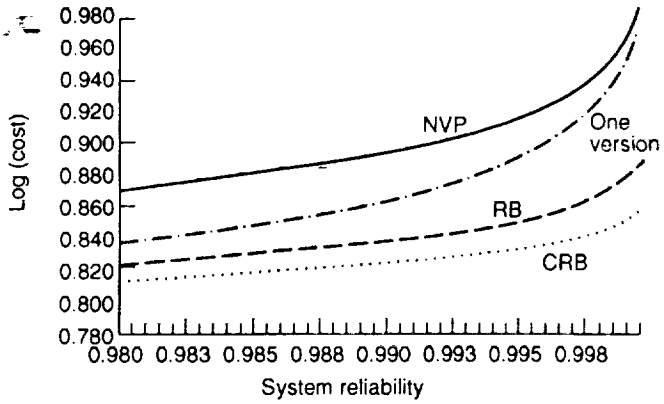


Figure 6. Optimal cost versus system reliability of each fault-tolerant technique considered, $\alpha = \beta = 1$

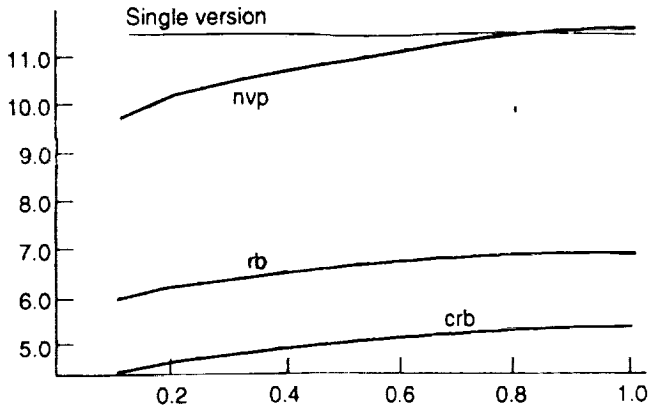


Figure 8. Log (cost) for R = 0.99999, $\alpha = 1$, $\beta_V = \beta_B = 0.1(0.1)1$

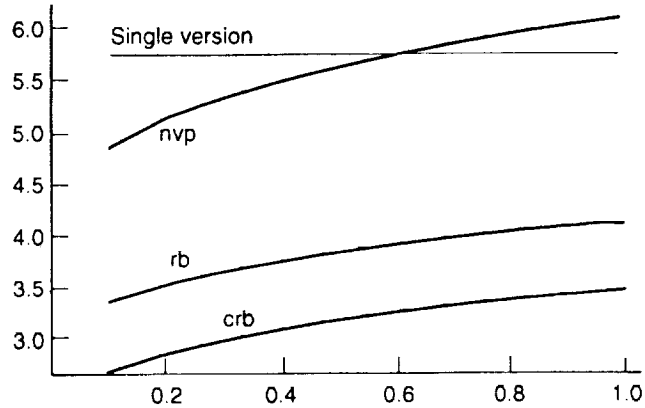


Figure 7. Log (cost) for R = 0.99999, $\alpha = 0.5$, $\beta_V = \beta_B = 0.1(0.1)1$

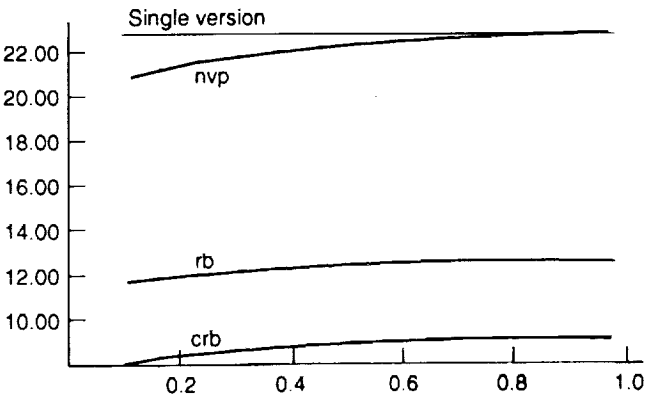


Figure 9. Log (cost) for R = 0.99999, $\alpha = 2$, $\beta_V = \beta_B = 0.1(0.1)1$

given system reliability R if all βs in the objective function are multiplied by the same constant k. Hence the problem is interesting only when the values of the βs are changed relative to each other for given values of the αs. To avoid getting lost in a morass of data the problem has been partitioned into three cases. In the first two cases the values of the βs for the versions are set to 1 and the optimal system costs calculated for the cases R = 0.9, 0.99, 0.999, 0.9999, and 0.99999 for various values of β_V and β_B. The Appendix, Tables 8–13, gives the results for $\beta_V = \beta_B = 0.1$ and 1 and $\alpha = 0.5$, 1, and 2.

Case 1

In case 1 both the β values for the acceptance test and the voter are equal: they were varied from 0.1 to 1.0 in steps of 0.1. These results are plotted for the case R = 0.99999 in Figures 7–9. In all cases, for a given system reliability R the cost of NVP > cost of RB > cost of CRB. As α increased, the difference between the three increased considerably, often by several orders of magnitude. For R = 0.99999 and $\alpha = 0.5$, CRB is an order of magnitude less costly than NVP. For $\alpha = 2$, the difference increases to over six orders of magnitude.

Also, for a given system reliability, $r_{NVP} > r_{RB} > r_{CRB}$, $V_{NVP} > V_{CRB}$, and $B_{RB} > B_{CRB}$. Except for low system reliabilities ($R = 0.9$), it was also the case for CRB that $B > V > r$. The acceptance test is considerably more critical than the voter or the versions.

RB and CRB were always less costly than a simplex system, which was not the case for NVP. As the voter became more costly to develop (β_V approached 1), the difference between a simplex system and NVP became more pronounced. Even if the voter and the acceptance test have the same development cost as the versions, it is always better to implement CRB than any of the other techniques.

It is clear that α has a more significant effect on cost than either β_B or β_V. Both RB and CRB are approximately loglinear for larger values of β_V and β_B less than 1.

Case 2

In case 2, as the CRB requires an acceptance test and a voter, to gain some intuition on the relationships between β_V and β_B they are varied from 0.25 to 1 in steps of 0.25. As in case 1 the βs for the versions are 1. In Figure 10 the cost is shown as a two-variable function of β_V and β_B for the case $\alpha = 1$ and $R = 0.99999$. The cases for $\alpha = 0.5$ and 2 are similar.

The function is monotone in both variables. The cost is slightly more affected by an increase in β_B.

The results for cases 1 and 2 were consistent among the cases considered.

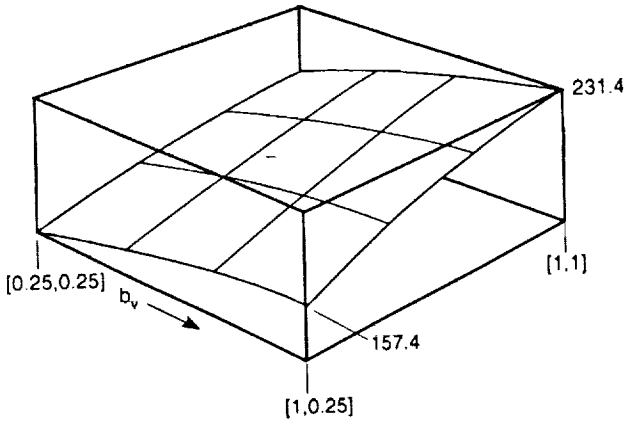


Figure 10. Plot of cost of CRB as function of β_1 and β_B for case $\alpha = 1$, $R = 0.99999$, β_B and $\beta_V = 0.25(0.25)1$

Table 6. Calculation of βs using minimum, maximum, and average costs

β	NVP	RB	CRB
$\sum\beta_i$	1.78, 2.25, 2.71	1.78, 2.37, 2.96	$RB\sum\beta_i + \beta_1$
β_1	1.00	1.00	1.00
β_V	0.001(≈0), 0.05, 0.1	–	0.001(≈0), 0.05, 0.01
β_B	–	0.001(≈0), 0.15, 0.3	0.001(≈0), 0.15, 0.3
$\beta_2 = \beta_3$	$1/2(\sum\beta_i - \beta_1 - \beta_1)$	$1/2(\sum\beta_i - \beta_1 - \beta_B)$	$1/2(\sum\beta_i - \beta_1 - \beta_1 - \beta_B)$

Case 3

In this case the range of costs proposed by Laprie *et al.*[3] was used for implementing NVP and RB. Their costs are expressed as the ratio of the fault-tolerant system cost to a simplex system cost for three environments: a maximum, a minimum, and an average cost. Furthermore, they have proposed that the cost of the voter in an NVP system varies from 0 to 0.1 of the cost of a single version. Similarly, the overhead for RB ranges from 0 to 0.3 of a single-version cost.

For each method of fault tolerance three runs were made using their maximum, minimum, and average costs. The values of the optimal solutions are given in Tables 14–16 in the Appendix. First, it was assumed that the βs in the model should sum to their ratio, then the costs were normalized to the primary or first version's cost by setting $\beta_1 = 1$. Table 6 shows how the other βs were calculated.

The values of the βs for the low, average, and high cost environments are given in Table 7. It was assumed that the cost of the second and third versions were the same and that for CRB, β_V was the same as for NVP and β_B was the same as for RB.

Considering Table 14 and Figure 11 (see Appendix),

Table 7. Values of β from Laprie *et al.*[3]

		βs		
		Low	Average	High
NVP				
	First version	1.000	1.000	1.000
	Other versions	0.390	0.600	0.805
	Voter	0.001	0.051	0.100
RB				
	First version	1.000	1.000	1.000
	Other versions	0.390	0.610	0.830
	Acceptance test	0.001	0.151	0.300
CRB				
	First version	1.000	1.000	1.000
	Other versions	0.390	0.610	0.830
	Voter	0.001	0.051	0.100
	Acceptance test	0.001	0.151	0.300

the minimum cost environment, it can be seen that in all cases except for the lowest system reliability requirement using NVP that a software fault-tolerant system is cheaper to implement than a simplex system with the same reliability. Also CRB is less expensive than RB, which is in turn cheaper than NVP. As the programming environment becomes more expensive, as in the average cost case, Table 15 and Figure 12 (see Appendix) show that the required system reliability must be greater than 0.95 for NVP to be cost competitive with a simplex system. The RB and CRB are always much cheaper than a simplex system. Similar results appear in Table 16 and Figure 13 (see Appendix) for NVP systems in a maximum cost environment. Also note that due to the high cost of the acceptance test, RB only becomes cost effective as the system reliability approaches 0.95. In all cases, the CRB provides the cheapest system reliability, even though five different software modules are required: three versions, a voter, and an acceptance test.

SUMMARY AND CONCLUSIONS

The above results have shown that in the case that failures are independent, consensus recovery block followed by recovery block are the most cost-justifiable fault-tolerant techniques to be considered. Unless the voter is perfect, N-version programming does not compete cost-wise with the other two methods. Indeed, in some cases it is worse than a simplex system. It is interesting to note that consensus recovery block, which contains both voting and recovery block, can provide considerable reduction in cost for a given system reliability over the other techniques, even when the cost of development of the voter and the acceptance test is the same as for the versions.

The authors are currently attempting to relax the condition that all version reliabilities are equal and that failures are independent. They intend to move to five- and seven-version systems to determine how costs are related.

ACKNOWLEDGEMENT

This research was supported in part by NASA grant NAG-1-983-1.

REFERENCES

1 **Saglietti, F and Ehrenberger, W** 'Software diversity – some considerations about its benefits and limitations' in *Proc. IFAC SAFECOMP '86* (1985) pp 27–34
2 **Laprie, J-C, Arlat, J, Beounes, C and Kanoun, K** 'Definition and analysis of hardware- and software-fault-tolerant architectures' *Computer* (July 1990) pp 39–51
3 **Scott, R K, Gault, J W, McAllister, D F and Wiggs, J** 'Experimental validation of six fault-tolerant software reliability models' *IEEE Fault Tolerant Comput. Syst.* Vol 14 (1984) pp 102–107
4 **Scott, R K, Gault, J W and McAllister, D F** 'The consensus recovery block' in *Proc. Total Systems Reliability Symp.* (December 1983) pp 74–85
5 **Scott, R K, Gault, J W and McAllister, D F** 'Modeling fault-tolerant software reliability' in *Proc. Third Symp. Reliability in Distributed Software and Database Systems* (October 1983) pp 15–27
6 **Scott, R K, and Gault, J W and McAllister, D F** 'Fault-tolerant software reliability modeling' *IEEE Trans. Soft. Eng.* Vol 13 No 5 (May 1987) pp 582–592
7 **Knight, J C and Leveson, N G** 'An experimental evaluation of the assumption of independence in multiversion programming' *IEEE Trans. Soft. Eng.* Vol 12 No 1 (January 1986)
8 **Kelly, J J P J, Eckhardt, D E, Caglayan, A *et al.*** 'Large scale second generation experiment in multi-version software: description and early results' *IEEE Fault Tolerant Comput. Syst.* Vol 18 (June 1988) pp 9–14
9 **Arlat, J, Kanoun, K and Laprie, J-C** 'Dependability modeling and evaluation of software fault-tolerant system' *IEEE Trans. Computers* Vol 39 No 4 (April 1990) pp 504–513
10 **Eckhardt, D E and Lee, L D** 'A theoretical basis for the analysis of multiversion software subject to coincident errors' *IEEE Trans. Soft. Eng.* Vol 11 No 12 (December 1985)
11 **Wolfram, S** *Mathematica* Addison-Wesley (1988)
12 **Vouk, M A** 'Back-to-back testing' *Inf. Soft. Technol.* Vol 32 No 1 (January–February 1990) pp 34–45
13 **Shoup, T E** *Numerical methods for the personal computer* Prentice Hall (1983) pp 64–69
14 **Vouk, M A, McAllister, D F, Caglayan, A *et al.*** 'Analysis of faults detected in a large-scale multi-version software development experiment' in *Proc. Digital Avionics Systems Conf.* (October 1990)
15 **Taylor, A E and Mann, W R** *Advanced calculus* (2nd ed) John Wiley (1972) pp 197–198
16 **Conte, S D and de Boor, C** *Elementary numerical analysis: an algorithmic approach* McGraw-Hill (1980)

APPENDIX

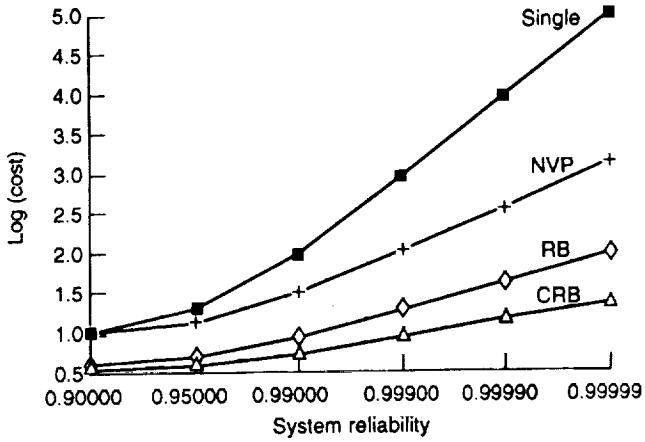


Figure 11. Plot of costs for minimum cost environment

Table 8. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	0.50000	0.82200	0.98229	0.89554	0.58268	0.96387	0.71350	0.50993	0.93884	0.80310	0.69154
0.99000	1.00000	0.94858	0.99764	1.18439	0.81147	0.98905	0.89570	0.71185	0.97204	0.92418	0.81624
0.99900	1.50000	0.98481	0.99968	1.47677	0.91483	0.99659	1.07887	0.82594	0.98627	0.96703	0.93424
0.99990	2.00000	0.99556	0.99996	1.78271	0.96181	0.99899	1.26714	0.89390	0.99323	0.98527	1.05114
0.99999	2.50000	0.99872	0.99999	2.10782	0.98307	0.99971	1.46212	0.93517	0.99672	0.99340	1.16907

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_V = 0.1$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_V = 0.5$

Table 9. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	0.50000	0.86367	0.94806	1.09735	0.69952	0.87230	0.91757	0.65491	0.81071	0.55906	0.94994
0.99000	1.00000	0.96368	0.99384	1.45455	0.87573	0.95772	1.12626	0.81259	0.90639	0.81011	1.09667
0.99900	1.50000	0.99016	0.99929	1.83078	0.94863	0.98644	1.33895	0.89334	0.95253	0.91649	1.23644
0.99990	2.00000	0.99737	0.99992	2.23250	0.97946	0.99594	1.56386	0.93865	0.97634	0.96359	1.37754
0.99999	2.50000	0.99931	0.99999	2.65873	0.99223	0.99882	1.79996	0.96479	0.98857	0.98427	1.52264

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_V = 1.0$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_V = 0.5$

Table 10. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	1.00000	0.83374	0.97165	1.33386	0.61578	0.93859	0.97480	0.53968	0.90186	0.78784	0.90350
0.99000	2.00000	0.95511	0.99584	1.95841	0.83342	0.97949	1.35956	0.73662	0.95220	0.90988	1.16412
0.99900	3.00000	0.98809	0.99942	2.62826	0.92833	0.99313	1.75139	0.84501	0.97545	0.95835	1.41212
0.99990	4.00000	0.99701	0.99993	3.37453	0.97007	0.99782	2.16486	0.90833	0.98742	0.98051	1.66093
0.99999	5.00000	0.99929	0.99999	4.20429	0.98804	0.99934	2.60454	0.94602	0.99368	0.99094	1.91617

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_1 = 0.1$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_1 = 1.0$

Table 11. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	1.00000	0.87379	0.94119	1.61039	0.72391	0.85161	1.24564	0.66635	0.77721	0.61989	1.20711
0.99000	2.00000	0.96971	0.99268	2.37218	0.89114	0.94788	1.66973	0.82220	0.88843	0.82888	1.50078
0.99900	3.00000	0.99287	0.99915	3.20399	0.95789	0.98211	2.10430	0.90155	0.94266	0.92081	1.78204
0.99990	4.00000	0.99838	0.99991	4.10399	0.98468	0.99420	2.56600	0.94549	0.97098	0.96360	2.06807
0.99999	5.00000	0.99998	0.99999	5.39217	0.99478	0.99817	3.04898	0.97029	0.98572	0.98345	2.36445

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_1 = 1.0$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_1 = 1.0$

Table 12. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	2.00000	0.85226	0.95646	2.27921	0.66513	0.90020	1.56577	0.58157	0.84795	0.76864	1.36788
0.99000	4.00000	0.96539	0.99349	3.68686	0.86413	0.96432	2.38211	0.76955	0.92293	0.89333	1.91442
0.99900	6.00000	0.99262	0.99916	5.29587	0.94614	0.98742	3.22177	0.86954	0.95946	0.94806	2.43804
0.99990	8.00000	0.99857	0.99991	7.10773	0.97995	0.99581	4.11933	0.92639	0.97889	0.97471	2.97057
0.99999	10.00000	0.99966	0.99999	8.71782	0.99305	0.99865	5.06931	0.95919	0.98929	0.98786	3.52524

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_1 = 0.1$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_1 = 2.0$

Table 13. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	2.00000	0.88586	0.93371	2.66075	0.74766	0.83054	1.91349	0.67738	0.74782	0.65793	1.72504
0.99000	4.00000	0.97652	0.99161	4.29364	0.90399	0.93840	2.77012	0.83108	0.87285	0.83906	2.31301
0.99900	6.00000	0.99548	0.99906	6.10758	0.96442	0.97811	3.64887	0.90877	0.93426	0.92203	2.87869
0.99990	8.00000	0.99917	0.99990	8.03590	0.98765	0.99265	4.58190	0.95121	0.96652	0.96252	3.45699
0.99999	10.00000	0.99976	0.99998	9.42361	0.99593	0.99762	5.55410	0.97457	0.98342	0.98225	4.05876

$\beta_1 = \beta_2 = \beta_3 = 1.0$ and $\beta_B = \beta_1 = 1.0$
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_B = \alpha_1 = 2.0$

Table 14. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	1.00000	0.80863	0.99541	0.97860	0.54795	0.99057	0.60677	0.45886	0.98463	0.95925	0.52878
0.99000	2.00000	0.94341	0.99924	1.51543	0.79237	0.99688	0.94908	0.67538	0.99270	0.98332	0.75437
0.99900	3.00000	0.98286	0.99987	2.04771	0.90479	0.99893	1.29284	0.80100	0.99623	0.99216	0.97023
0.99990	4.00000	0.99487	0.99998	2.59536	0.95658	0.99964	1.64132	0.87699	0.99803	0.99618	1.18297
0.99999	5.00000	0.99816	0.99999	3.04677	0.98036	0.99989	1.99720	0.92375	0.99897	0.99812	1.39535

Low cost environment

Table 15. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	1.00000	0.82948	0.97541	1.17540	0.64270	0.91782	0.90583	0.56042	0.85763	0.84523	0.80891
0.99000	2.00000	0.95329	0.99632	1.78498	0.84860	0.97233	1.30363	0.75034	0.92910	0.93461	1.07194
0.99900	3.00000	0.98737	0.99947	2.43328	0.93648	0.99075	1.70983	0.85397	0.96282	0.97016	1.32165
0.99990	4.00000	0.99676	0.99993	3.15311	0.97437	0.99708	2.14078	0.91420	0.98053	0.98626	1.57218
0.99999	5.00000	0.99926	0.99999	3.90712	0.99023	0.99911	2.59855	0.94986	0.99000	0.99373	1.82932

Average cost environment

Table 16. Component reliability and log (cost)

One version		NVP			RB			CRB			
r	Cost	r	V	Cost	r	B	Cost	r	B	V	Cost
0.90000	1.00000	0.83552	0.97010	1.28359	0.66553	0.89989	1.03939	0.58549	0.82471	0.82487	0.93951
0.99000	2.00000	0.95586	0.99565	1.91439	0.86104	0.96598	1.44654	0.76755	0.91192	0.92538	1.20922
0.99900	3.00000	0.98837	0.99940	2.59301	0.94296	0.98859	1.86293	0.86547	0.95365	0.96604	1.46523
0.99990	4.00000	0.99707	0.99992	3.31928	0.97765	0.99639	2.30558	0.92183	0.97572	0.98447	1.72282
0.99999	5.00000	0.99907	0.99998	3.93675	0.99180	0.99889	2.77405	0.95489	0.98755	0.99297	1.98810

High cost environment

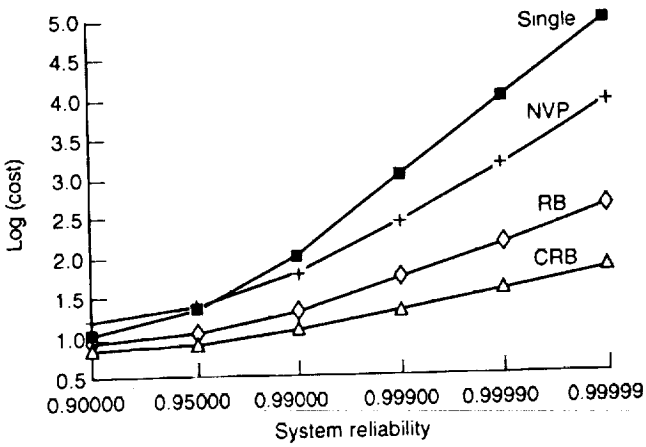


Figure 12. Plot of costs as programming environment becomes more expensive

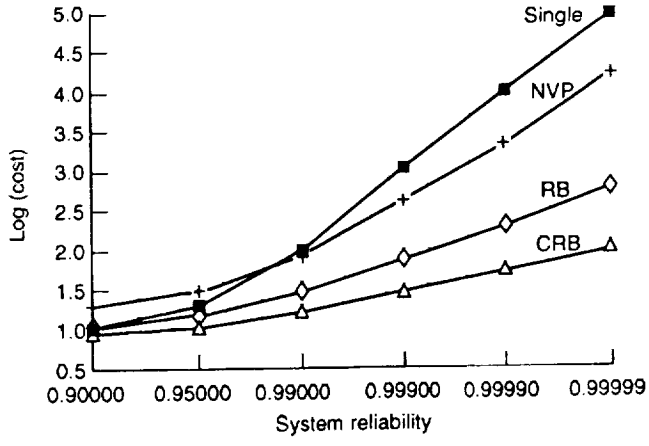


Figure 13. Plot of costs for maximum cost environment